DEA : Représentation de la Connaissance et Formalisation du Raisonnement
Responsable : Claudette Cayrol
2002–2003

Application de la résolution de conflits "logiques", aux conflits des problèmes d'ordonnancement

par Pascal Lafourcade

Responsables de stage :
Hélène Fargier, Marie-Christine Lagasquie-Schiex

Mots-clés : Intelligence artificielle, problème d'ordonnancement, résolution de conflits, aide à la décision, logique propositionnelle.

Résumé :
Lorsqu'il n'existe pas de solution à un problème d'ordonnancement, le calcul des *conflits*, ensembles minimaux de contraintes "incohérents", donne les explications de cet échec. Cette notion de *conflits* existe aussi en logique propositionnelle (ensemble minimaux de formules inconsistantes). Nous allons adapter de nombreux critères de préférences locales issus de la résolution de conflits en logique propositionnelle, à la résolution de conflits dans un problème d'ordonnancement. Nous introduisons également de nouveaux critères "colorés" spécifiques aux problèmes d'ordonnancement. Ce cadre formel d'aide à la décision permet de transformer un problème d'ordonnancement sans solution en un problème d'ordonnancement avec solution, en respectant les préférences locales. Nous spécifions pour tous nos critères des algorithmes de type "Branch-and-Bound" pour la recherche de solutions optimales. L'implémentation de certains critères nous montre que l'ordre de résolution des conflits est crucial et confirme que la résolution de tels conflits est un problème exponentiel.

Remerciements

Je remercie Denis Souloumiac, Président du Basket Labège Auzeville Club, de m'avoir permis de m'inscrire à ce DEA tout en continuant à travailler en tant qu'en emploi jeune au sein du club.

Je remercie Claudette Cayrol, Professeur, de m'avoir accueilli au sein de sa formation doctorale et de son équipe RPDMP à l'IRIT.

Je tiens à remercier Marie-Christine Lagasquie-Schiex, Maître de Conférences, et Hélène Fargier, chargée de recherche CNRS, de m'avoir proposé ce sujet de stage, ainsi que pour la formation et les conseils qu'elles m'ont prodigués au cours de nos diverses discussions.

J'exprime ma gratitude à Claudette Cayrol, Professeur, pour la rigueur qu'elle m'a apportée. Ses conseils et ses remarques m'ont permis de faire avancer ce rapport.

Je remercie également Jérôme Lang et Hélène Fargier pour m'avoir accueilli dans leur bureau, durant mon stage.

Je remercie David, Frédéric et Thierry pour avoir partagé les repas quotidiens, et enfin tout le staff technique, Maïté, Katia et Michel, qui m'a soutenu durant cette année et qui me supporte depuis plus longtemps encore.

Table des matières

Introduction

Dans l'ordonnancement de tâches, ou gestion de projet, une tâche possède une date de début, une durée, une date de disponibilité, une date de livraison et des contraintes de précédence. Pour résoudre un tel problème il faut trouver une date de début pour toutes les tâches du problème. De nombreux travaux [KP] existent pour trouver une solution à un problème d'ordonnancement et même une solution optimale suivant les préférences de l'utilisateur *a priori*. Lorsqu'il n'existe pas de solution, il apparaît des incompatibilités entre les contraintes des tâches. Ces incompatibilités sont appelées *conflits* : ensembles minimaux de contraintes ne pouvant être satisfaites simultanément. Dans [FGT02], les auteurs proposent une méthode pour identifier les conflits dans les graphes de tâches, basée sur la propagation des dates au plus tôt (date avant laquelle une tâche ne peut débuter).

En logique propositionnelle apparaît également le terme de conflit, c'est un ensemble minimal inconsistant de formules logiques. Il existe des critères permettant d'aider l'utilisateur à rétablir la consistance d'un ensemble de formules inconsistantes, à l'aide de préférences [CLSM00].

Une fois les conflits identifiés, il s'agit de les "résoudre" en prenant en compte les préférences de l'utilisateur. Lorsque ces préférences sont globales, *i.e.* l'utilisateur donne un ordre global de préférence des tâches *a priori*, il existe des méthodes issues des CSP pour trouver la solution optimales suivant ces préférences. Nous allons nous intéresser à des préférences locales. L'utilisateur va nous indiquer pour chaque conflit identifié quelles contraintes il peut éventuellement modifier ou pas. Nous interprétons ces préférences locales par une coloration locale des contraintes des conflits en 3 couleurs (rouge, bleu, vert). En étudiant d'une part les critères d'aide à la résolution de conflits en logique propositionnelle, et d'autre part les conflits dans un problème d'ordonnancement, nous introduisons de nouveaux critères basés sur cette coloration locale. Ainsi, en fonction de ces préférences locales, nous proposons une solution : ensemble de modifications à apporter au problème d'ordonnancement pour qu'il devienne "consistant". Les méthodes de résolution de la logique propositionnelle, algorithme de type "Davis et Putnam" [DP60], utilisés par Castell [Cas97] et Leberre [Leb00] vont être adaptées pour chacun de nos critères locaux pour résoudre les problèmes d'ordonnancement. Ces algorithmes seront améliorées par des coupures de branche dans l'arbre de recherche, technique dite de "Branch-and-Bound" [LW66]. Nous implémentons et testons sur des problèmes de conflits produits par par Kolisch et Sprecher dans [KS97], ces différents algorithmes.

Nous organisons notre travail de la manière suivante. Dans le chapitre 1, nous définissons et identifions les "conflits" dans les problèmes d'ordonnancement de tâches afin de les résoudre. Dans le chapitre 2, nous étudions les différents critères de préférence colorés ou non, pour la résolution des conflits d'un point de vue "logique". Dans le chapitre 3, nous adaptons les méthodes de résolution des conflits "logiques", aux conflits obtenus dans un problème d'ordonnancement. Dans le chapitre 4, en nous inspirant de la méthode de construction des conflits nous adaptons les algorithmes de type "Branch-and-Bound" aux critères choisis, pour prendre en compte la spécificité du problème. Nous comparons les résultats des algorithmes implémentés sur des jeux de tests, pour la recherche de toutes les solutions et des solutions parcimonieuses, avec deux heuristiques.

Chapitre 1

Formalisation du Pert

1.1 Définitions, représentations

1.1.1 Définition

Un problème d'ordonnancement est constitué d'un ensemble fini de tâches soumises à des contraintes de temps. [EL99]

Définition 1 (Problème d'ordonnancement) *Soit un ensemble de tâches $T = \{x_1, \dots, x_n\}$, à chaque tâche $x_i \in T$ est associé un quadruplet (t_i, d_i, r_i, l_i), où*
- $t_i \geq 0$, *représente la date de début de la tâche x_i.*
- $d_i \geq 0$, *représente la durée de la tâche x_i.*
- $l_i \geq 0$, *représente la date de livraison de la tâche, i.e. la tâche x_i doit se finir avant la date l_i.*
- $r_i \geq 0$, *représente la date de disponibilité, i.e. la tâche x_i ne peut commencer qu'après la date r_i.*

Seuls les t_i sont des variables.
Trois ensembles de contraintes S, R et L lient ces données :
- *S ensemble de contraintes de précédence, de la forme $t_j \geq t_i + d_i$, i.e. la tâche x_j doit être réalisée après la tâche x_i.*
- *R ensemble de contraintes de disponibilité, de la forme $t_i \geq r_i$, i.e. la tâche x_i ne peut commencer qu'après la date r_i.*
- *L ensemble de contraintes de livraison, de la forme $t_i + d_i \leq l_i$, i.e. la tâche x_i doit se finir avant la date l_i.*

Un problème d'ordonnancement est donc un quadruplet $< T, S, R, L >$.

Nous introduisons une tâche initiale de durée nulle, notée α, la première tâche du projet qui précède toutes tâches (sans prédécesseur). Nous introduisons également une tâche finale de durée nulle, notée ω, qui succède à toutes les tâches (sans successeur).

Exemple 1 *Pour construire une maison, il faut construire les murs, mettre la toiture, couler les fondations, poser les cloisons, installer l'électricité, l'eau, le gaz, faire les plâtres, les peintures, crépir ... Toutes ces tâches prennent un certain temps et doivent être effectuées dans un certain ordre. Le problème d'ordonnancement de la construction d'une maison peut s'exprimer de la manière suivante :*
- d_{murs} = *3 jours. % durée de la tâche "construire les murs" 3 jours %*
- $d_{toiture}$ = *8 jours.*
- $d_{fondation}$ = *5 jours.*
- $d_{cloisons}$ = *3 jours.*
- $d_{électricité}$ = *4 jours.*
- d_{eau} = *5 jours.*
- d_{gaz} = *3 jours.*
- $d_{plâtre}$ = *6 jours.*
- $d_{peinture}$ = *4 jours.*
- $d_{crépi}$ = *5 jours.*
- d_{fin} = *0 jour.*

Sachant en plus que nous avons certaines contraintes de précédence, les fondations doivent être réalisées en premier, puis les murs, ensuite la toiture, puis les cloisons et le crépi. Après les cloisons, peuvent débuter les travaux sur l'eau, le gaz et l'électricité enfin les plâtres et la peinture. Nous obtenons les contraintes de précédence suivantes :

- $t_{fondation} + 5 \leq t_{murs}$ *% le début des fondations + durée des fondations ≤ début de la construction des murs %*
- $t_{murs} + 3 \leq t_{toiture}$
- $t_{toiture} + 8 \leq t_{cloisons}$
- $t_{toiture} + 8 \leq t_{crépi}$
- $t_{cloisons} + 3 \leq t_{eau}$
- $t_{cloisons} + 3 \leq t_{gaz}$
- $t_{cloisons} + 3 \leq t_{électricité}$
- $t_{électricité} + 4 \leq t_{plâtre}$
- $t_{gaz} + 3 \leq t_{plâtre}$
- $t_{eau} + 5 \leq t_{plâtre}$
- $t_{plâtre} + 6 \leq t_{peinture}$
- $t_{peinture} + 4 \leq t_{fin}$
- $t_{crépi} + 5 \leq t_{fin}$

Les différents artisans "imposent" des dates de disponibilité pour la réalisation de la maison, ce qui risque de retarder les travaux. Par exemple, le plâtrier n'est disponible qu'à partir du 15 novembre, ce qui implique : $t_{plâtre} \geq 15/11$. *Le peintre doit finir avant le 20 Décembre cela se traduit par :* $t_{peinture} + d_{peinture} \leq 20/12$.

Comme le montre ce petit exemple, les problèmes d'ordonnancement deviennent vite très difficiles à résoudre. De nombreuses méthodes existent pour résoudre de tels problèmes ([GNZ86, BKSO96, KP, SFV95, EGJ02]). Pour une bonne appréhension globale, ces problèmes sont souvent modélisés par des graphes orientés.

1.1.2 Représentation d'un problème d'ordonnancement par un graphe

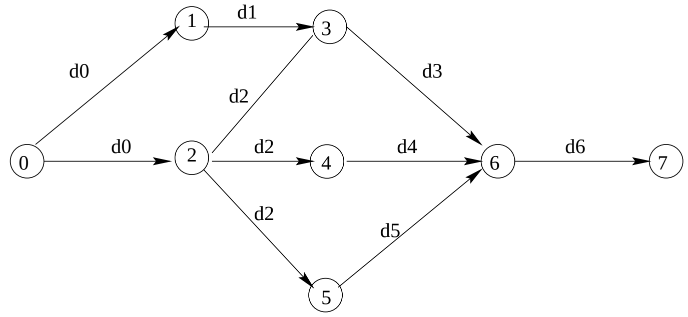
Soit $T = \{x_1, \ldots, x_n\}$ un ensemble de tâches. Représentons les contraintes de précédence d'un problème d'ordonnancement de tâches par un graphe orienté $G = (X, U)$. On suppose que les contraintes de précédence sont cohérentes entre elles, *i.e.* le graphe orienté associé sera sans circuit (annexe C page 53 : Rappels sur les graphes orientés).

Il existe deux représentations possibles sous forme de graphe orienté d'un problème d'ordonnancement décrites dans [GM85].

1.1.2.1 Graphe potentiels-tâches

Dans la représentation potentiels-tâches développée par B. Roy dans [Roy60] :
- Chaque tâche $x_i \in T$ est représentée par un sommet du graphe.
- Il existe un arc orienté de la tâche x_i vers la tâche x_j si la tâche x_i précède la tâche x_j. Cet arc est étiqueté par la durée d_i de la tâche x_i.

Exemple 2 *Graphe potentiels-tâches.*



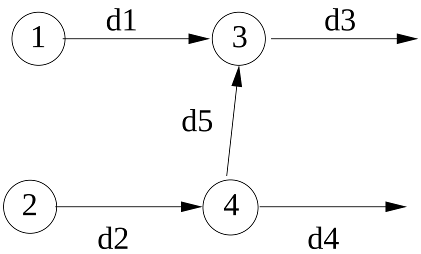
1.1.2.2 Graphe potentiels-étapes

Dans la représentation potentiels-étapes exprimée par Malcom, Roseboom, Clark et Farzar dans [MRCF59] :

- Les sommets sont les débuts et fins des étapes du projet.
- Chaque arc représente une tâche $x_i \in T$.
- Chaque arc représente également une contrainte de précédence *i.e.* si la tâche x_i précède la tâche x_j alors la fin de l'arc associé à x_i rejoint le sommet symbolisant le début de x_j.
- Chaque arc entre deux sommets est valué par la durée d_i de sa tâche.

Remarque 1 *Dans une telle représentation, pour exprimer certaines contraintes de précédence nous devons introduire des tâches fictives de durée nulle.*

Exemple 3 *La tâche* 1 *précède la tâche* 3.
La tâche 2 *précède la tâche* 3 *et la tâche* 4.
Comme les tâches sont représentées par les arcs, elles imposent l'ajout de la tâche 5 "fictive" de "précédence" entre 4 et 3 de durée nulle.

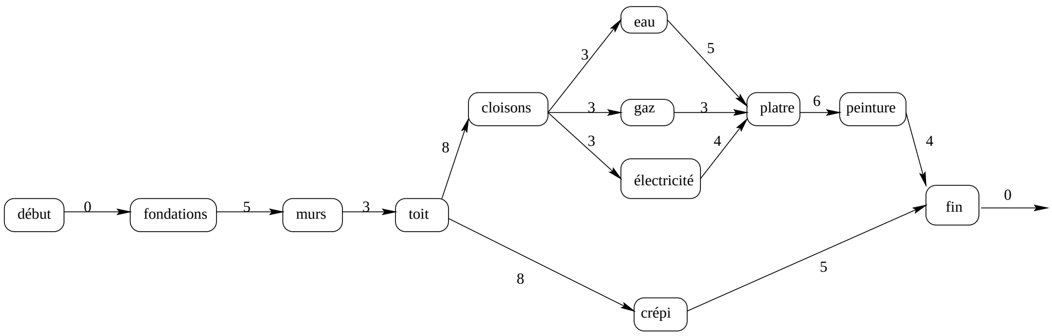


Remarque 2 *Les problèmes d'ordonnancement, représentés par un graphe potentiels-étapes ou potentiels-tâches, sont généralement appelés PERT (Program Evaluation and Review Technique).*[1]

1.1.2.3 Conclusion

Nous choisirons la représentation "potentiels-tâches" car elle n'oblige pas à rajouter des tâches fictives.

Exemple 4 *Représentons l'exemple 1 page 3 par son graphe potentiels-tâches, seules les contraintes de précédence et les durées sont représentées.*

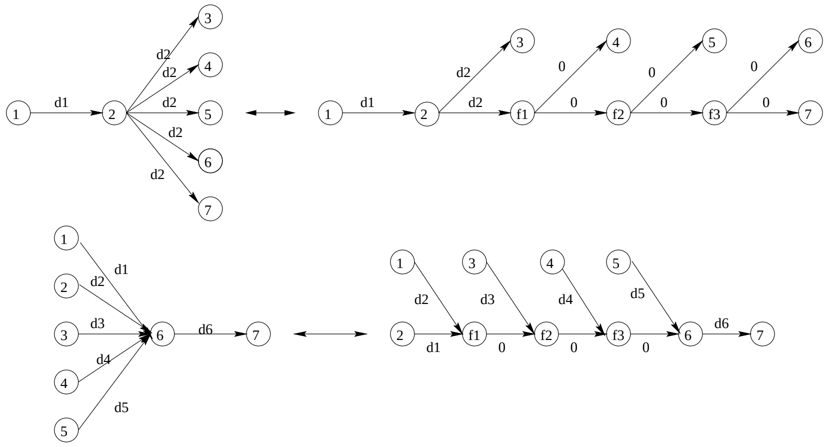


Dans la suite un problème d'ordonnancement P sera représenté par un graphe potentiels-tâches (T, S).

La création d'un graphe de tâches permet de mieux visualiser les différentes étapes d'un problème d'ordonnancement.

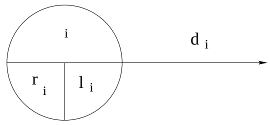
Remarque 3 *Dans la représentation potentiels-tâches, tout graphe peut être "binarisé" quitte à ajouter des tâches fictives, i.e. chaque nœud ne possède qu'un successeur et au plus deux prédécesseurs ou chaque nœud ne possède qu'un prédécesseur et au plus deux successeurs. Les tâches* $f1, f2, f3$ *sont fictives de durée nulle.*

[1] Ce terme provient du nom d'un projet développé par la Défense Américaine, adapté au problème d'ordonnancement.



Grâce à la remarque 3 page précédente, nous ne considérerons désormais que des exemples où les graphes sont “binarisés” sans perdre en généralité.

Notation : On représentera sur les graphes une tâche par son numéro, par sa date de disponibilité, par sa date de livraison et par sa durée.



Il faut maintenant savoir si toutes les contraintes, ainsi exprimées, peuvent être satisfaites. Est-il possible de trouver une date de début pour toutes les tâches, sans avoir d'incompatibilité entre les contraintes de $S \cup R \cup L$?

1.2 Consistance et conflit dans un problème d'ordonnancement

Un problème d'ordonnancement est-il réalisable ? Est-il consistant ?

Définition 2 (Problème d'ordonnancement consistant) *Un problème d'ordonnancement de tâches $P =< T, S, R, L >$ est consistant si et seulement s'il existe une assignation de tous les t_i, satisfaisant l'ensemble des contraintes du problème i.e. $S \cup L \cup R$. Sinon le problème sera dit inconsistant.*

Caractérisons et localisons de manière plus précise les conflits.

Définition 3 (Chemin dans un problème d'ordonnancement de tâches) *Soit $P =< T, S, R, L >$ un problème d'ordonnancement de tâches, un chemin correspond à un chemin élémentaire (cf annexe C page 53) du graphe associé à P, partant de la tâche x_i allant jusqu'à la tâche x_j et regroupant toutes les contraintes de S, R, L où les tâches $x_i, \dots, x_j$ apparaissent; ce chemin sera noté $(x_i, \dots, x_j)$.*

Remarque 4 *Un chemin représente donc toutes les durées, date de disponibilité, de livraison des tâches présentes dans le chemin, plus toutes les inéquations associées à ces contraintes et les inéquations de précédence permettant de relier les différentes tâches entre elles. La dernière tâche x_j d'un chemin $(x_i, \dots, x_j)$ d'un problème d'ordonnancement de tâches est accomplie i.e. il ne faut pas oublier de prendre en compte sa durée.*

Définition 4 (Chemin consistant, critique et inconsistant) *Un chemin $(x_i, \dots, x_{i+l})$ sera dit :*
– consistant si et seulement si le temps dont on dispose de x_i à x_{i+l} est suffisant par rapport au temps que l'on va mettre pour réaliser les tâches.

$$\sum_{k, x_k \in (x_i, \dots, x_{i+l})} d_k < l_{i+l} - r_i$$

– *critique si et seulement si le temps dont on dispose de x_i à x_{i+l} est juste par rapport au temps que l'on va mettre pour réaliser les tâches.*

$$\sum_{k, x_k \in (x_i, \dots, x_{i+l})} d_k = l_{i+l} - r_i$$

– inconsistant si et seulement si le temps dont on dispose de x_i à x_{i+l} est insuffisant par rapport au temps que l'on va mettre pour réaliser les tâches.

$$\sum_{k, x_k \in (x_i, \dots, x_{i+l})} d_k > l_{i+l} - r_i$$

Remarque 5 *Un chemin consistant (respectivement critique et inconsistant) réduit à une seule tâche sera appelé par abus de langage tâche consistante (respectivement critique et inconsistante).*

Notation : $prec(x_i)$ et $succ(x_i)$ désignent les prédécesseurs et les successeurs de x_i. (cf Annexe C définition 53 page 53).

Remarque 6 *D'après l'algorithme 9 page 54 en annexe C, on peut numéroter les tâches d'un problème d'ordonnancement de telle sorte que si le sommet x_i précède le sommet x_j dans le graphe alors le numéro associé à x_i est strictement inférieur à celui associé à x_j.*

Une tâche x_i ne peut débuter que lorsque toutes les tâches qui y aboutissent sont terminées. La notion de date de début au plus tôt représente la plus petite date à partir de laquelle la tâche pourra être effectuée.

Définition 5 (Date de début au plus tôt)
Soit un problème d'ordonnancement de tâches $P =< T, S, R, L >$ représenté par le graphe $G = (T, S)$. Pour tout sommet $x_i \in T$, $\underline{t}_i$ sa date de début au plus tôt, est définie par :

$$\underline{t}_i = \max(r_i, \max_{x_j \in prec(x_i)} (\underline{t}_j + d_j))$$

L'algorithme 1 permet de calculer par propagation avant les dates de début au plus tôt de toutes les tâches, à condition de travailler avec des tâches numérotées comme dans la remarque 6. Cet algorithme a une complexité de $O(m + n)$ où $n =$ nombre de tâches et $m =$nombre d'arcs [GJ79, Pap94].

Algorithme 1: Calcul des dates au plus tôt

début
 % Initialisation de la date au plus tôt avec la disponibilité associée à la tâche %
 pour *toutes les tâches x_i* **faire**
  $plustot[x_i] = max(0, r[x_i])$;
 % Calcul des dates au plus tôt par propagation avant%
 pour *toutes les tâches x_i de $tache1$ à $tachen$* **faire**
  pour *toutes les tâches $tachej$ qui succèdent à x_i* **faire**
   si $plustot[x_i] + d[x_i] > plustot[tachej]$ **alors**
    $plustot[tachej] = plustot[x_i] + d[x_i]$;
fin

Nous introduisons la date de début au plus tard, date à laquelle peut commencer une tâche sans remettre en cause la date de fin du projet.

Définition 6 (Date de début au plus tard)
Soit un problème d'ordonnancement de tâches $P =< T, S, R, L >$ représenté par le graphe $G = (X, U)$. Pour tout sommet $x_i \in X$ on définit $\overline{t}_i$ sa date de début au plus tard :

$$\overline{t}_i = \min(l_i - d_i, \min_{x_j \in succ(x_i)} (\overline{t}_j - d_i))$$

L'algorithme 2 permet de calculer par propagation arrière les dates de début au plus tôt de toutes les tâches, à condition de travailler avec des tâches numérotées comme dans la remarque 6 page précédente. Cet algorithme a une complexité de $O(m+n)$ où n = nombre de tâches et m = nombre d'arcs [GJ79, Pap94].

Algorithme 2: Calcul des dates au plus tard

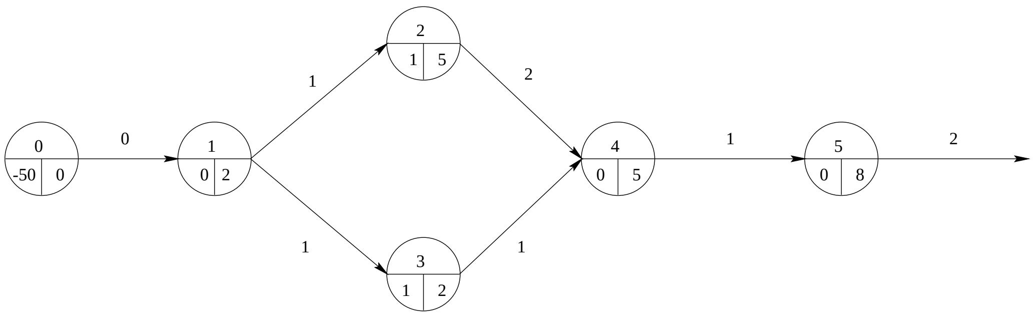
début
- % Initialisation de la date au plus tard avec la livraison moins la durée associée à la tâche %
- **pour** *toutes les tâches* x_i **faire**
  - $plustard[x_i] = l[x_i] - d[x_i]$;
- % calcul des dates au plus tard par propagation arrière %
- **pour** *toutes les tâches* x_i *de* $tachen$ *à* $tache1$ **faire**
  - **pour** *toutes les tâches* $tachej$ *qui précédent* x_i **faire**
    - **si** $plustard[tachej] > plustard[x_i] - d[tachej]$ **alors**
      - $plustard[tachej] = plustard[x_i] - d[tachej]$;

fin

Définition 7 (Criticité d'une tâche) *La criticité d'une tâche x_i est la différence entre sa date au plus tôt et sa date au plus tard.*

$$\textit{Criticité } (x_i) = \overline{t}_i - \underline{t}_i$$

Exemple 5 (calcul des dates de début au plus tôt et au plus tard) *Le problème d'ordonnancement est entièrement représenté par son graphe de précédence.*



Calcul des dates au plus tôt des toutes les tâches.
$\underline{t}_0 = -50$
$\underline{t}_1 = 0$
$\underline{t}_2 = max(1, 0+1) = 1$
$\underline{t}_3 = max(1, 0+1) = 1$
$\underline{t}_4 = max(0, max(1+2, 1+1)) = 3$
$\underline{t}_5 = max(0, 3+1) = 4$

Calcul des dates au plus tard des toutes les tâches.
$\overline{t}_5 = 8 - 2 = 6$
$\overline{t}_4 = min(5-1, 6-1) = 4$
$\overline{t}_3 = min(2-1, 4-1) = 1$
$\overline{t}_2 = min(5-2, 4-2) = 2$
$\overline{t}_1 = min(2-1, min(1-1, 2-1)) = 0$

$\overline{t}_0 = min(0,0) = 0$

On obtient pour chaque tâche les intervalles [date de début au plus tôt,date de début au plus tard] suivants :
En x_0 *:* $[-50, 0]$ *tâche consistante*
En x_1 *:* $[0, 0]$ *tâche critique*
En x_2 *:* $[1, 2]$ *tâche consistante*
En x_3 *:* $[1, 1]$ *tâche critique*
En x_4 *:* $[3, 4]$ *tâche consistante*
En x_5 *:* $[4, 6]$ *tâche consistante*

Passons la date de livraison de la tâche 5 à 5 au lieu de 8. Les dates au plus tôt ne changent pas.
$\underline{t}_0 = -50$
$\underline{t}_1 = 0$
$\underline{t}_2 = 1$
$\underline{t}_3 = 1$
$\underline{t}_4 = 3$
$\underline{t}_5 = 4$

Par contre les dates au plus tard sont modifiées.
$\overline{t}_5 = 3$
$\overline{t}_4 = 2$
$\overline{t}_3 = 1$
$\overline{t}_2 = 0$
$\overline{t}_1 = -1$
$\overline{t}_0 = -1$

On obtient pour chaque tâche les intervalles [date de début au plus tôt,date de début au plus tard] suivants :
En x_0 *:* $[-50, -1]$ *tâche consistante*
En x_1 *:* $[0, -1]$ *tâche inconsistante*
En x_2 *:* $[1, 0]$ *tâche inconsistante*
En x_3 *:* $[1, 1]$ *tâche critique*
En x_4 *:* $[3, 2]$ *tâche inconsistante*
En x_5 *:* $[4, 3]$ *tâche inconsistante*

Théorème 1 *[FGT02] Un problème d'ordonnancement de tâches* $P =< T, S, R, L >$ *est inconsistant si et seulement s'il existe un chemin inconsistant.*

Preuve :
$\Leftarrow$ Supposons qu'il existe un chemin $(x_i, \dots, x_{i+l})$ inconsistant. Regardons certaines des contraintes associées à ce chemin :
- $t_i + d_i \leq t_{i+1}$
- $t_{i+1} + d_{i+1} \leq t_{i+2}$
- ...
- $t_{i+l-1} + d_{i+l-1} \leq t_{i+l}$

Sommons membre à membre les inéquations ci-dessus, on obtient :

$$t_i + \sum_{k, x_k \in x_i, \dots, x_{i+l-1}} d_k \leq t_{i+l}$$

Rajoutons la contrainte de disponibilité de la tâche x_i *i.e.* $t_i \geq r_i$ et la contrainte de livraison de la tâche x_{i+l} *i.e.* $t_{i+l} + d_{i+l} \leq l_{i+l}$. On obtient alors

$$\sum_{k, x_k \in x_i, \dots, x_{i+l}} d_k + r_i \leq l_{i+l}$$

Or le chemin $(x_i, \ldots, x_{i+l})$ est inconsistant *i.e.* par définition que

$$\sum_{k, x_k \in x_i, \ldots, x_{i+l}} d_k + r_i > l_{i+l}$$

Il y a une contradiction quelle que soit l'assignation pour les t_i, donc il n'existe pas d'assignation des t_i pour le problème d'ordonnancement P, donc il est inconsistant.

$\Rightarrow$ Pour montrer que, si un problème est inconsistant alors il existe un chemin inconsistant, nous allons raisonner par contraposée. Supposons qu'il n'existe pas de chemin inconsistant et montrons que le problème est consistant. Tous les chemins sont alors critiques ou consistants. Partant de la tâche α nous calculons les dates de début au plus tôt, de toutes les tâches par propagation avant dans le graphe. Par propagation arrière à partir de ω nous calculons les dates de début au plus tard de toutes les tâches. Pour chaque tâche x_i , montrons que $\underline{t}_i \leq t_i$ et que $t_i \leq \overline{t}_i$ ainsi $[\underline{t}_i, \overline{t}_i]$ forment un intervalle dans lequel nous pouvons choisir une valeur pour t_i.

Par induction sur le nombre de tâches d'un problème d'ordonnancement montrons que pour toutes les tâches x_i on a $\underline{t}_i \leq t_i$ et $t_i \leq \overline{t}_i$.

- Soit P un problème à une seule tâche x_i, où "tous les chemins sont consistants".

$$\underline{t}_i = \max(r_i, \max_{x_j \in prec(x_i)} (\underline{t}_j + d_j))$$

  x_i n'a pas de prédécesseur donc $\underline{t}_i = r_i$; de plus la contrainte de disponibilité $r_i \leq t_i$ prouve que $\underline{t}_i \leq t_i$.

$$\overline{t}_i = \min(l_i - d_i, \min_{x_j \in succ(x_i)} (\overline{t}_j - d_i))$$

  x_i n'a pas de successeur donc $\overline{t}_i = l_i - d_i$; de plus la contrainte de précédence $t_i + d_i \leq l_i$ prouve que $t_i \leq \overline{t}_i$.
  La consistance du chemin assure que $\underline{t}_i \leq t_i$ et $t_i \leq \overline{t}_i$ existent simultanément.

- Soit P un problème à deux tâches x_i et x_j avec x_i qui précède x_j, où tous les chemins sont consistants.
  - La tâche x_i n'a pas de prédécesseur donc on a $\underline{t}_i \leq t_i$ (voir cas précédent).
  - Pour la tâche x_j, nous envisageons deux cas :
    - Si $\underline{t}_j = rj$ avec la contrainte de disponibilité $r_j \leq t_j$ on prouve que $\underline{t}_j \leq t_j$.
    - Sinon

$$\underline{t}_j = \max_{x_k \in prec(x_j)} (\underline{t}_k + d_k)$$

    donc $\underline{t}_j = \underline{t}_i + d_i$; la contrainte de précédence $t_i + d_i \leq t_j$ et $\underline{t}_i \leq t_i$ nous assure que $\underline{t}_i + d_i \leq t_j$ donc que $\underline{t}_j \leq t_j$.
  - La tâche x_j n'a pas de successeur donc on a donc $t_j \leq \overline{t}_j$ (voir cas précédent).
  - Pour la tâche x_i, nous envisageons deux cas :
    - Si $\overline{t}_i = l_i - d_i$ avec la contrainte de précédence $t_i + d_i \leq l_i$ on prouve que $t_i \leq \overline{t}_i$.
    - Sinon

$$\overline{t}_i = \min_{x_k \in succc(x_i)} (\overline{t}_k - d_i)$$

    Donc $\overline{t}_i = \overline{t}_j - d_i$, comme $t_j \leq \overline{t}_j$ on obtient $t_j - d_i \leq \overline{t}_j - d_i = \overline{t}_i$. De plus la contrainte de précédence $t_i \leq t_j - d_i$ nous assure que $t_i \leq \overline{t}_i$.

  La consistance du chemin assure que $\underline{t}_i \leq t_i$, $t_i \leq \overline{t}_i$ et $\underline{t}_j \leq t_j$, $t_j \leq \overline{t}_j$ existent simultanément.

  En remontant s'il le faut jusqu'à la tâche initiale α pour obtenir $\underline{t}_i \leq t_i$ et en propageant jusqu'à ω on obtient $t_i \leq \overline{t}_i$ pour toutes les tâches simultanément grâce à la consistance de tous les chemins.

Donc le problème est consistant.

□

Définition 8 (Sous-problème) *[FGT02]*
Soit $P =< T, S, R, L >$ un problème d'ordonnancement, $P' =< T', S', R', L' >$ est un sous-problème de P si $T' \subseteq T$,$S' \subseteq S$,$R' \subseteq R$, $L' \subseteq L$ et $P' \neq P$.

Définition 9 (Conflit) *[FGT02]*
Un sous-problème P' inconsistant d'un problème P d'ordonnancement dont tous les sous-problèmes sont consistants (minimalité) est appelé conflit de P.

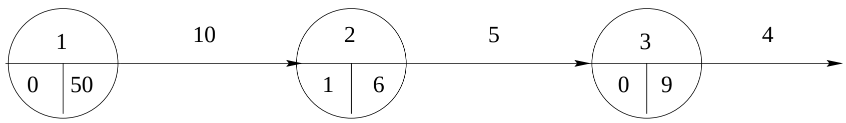
Propriété 1 (Caractérisation d'un conflit) *[FGT02]*
Soit P un problème d'ordonnancement. $P' =< T', S', R', L' >$ est un conflit de P entre les tâches x_i et x_j si et seulement si P' est représenté par un chemin $(x_i, \ldots, x_j)$ dont :
- *S' l'ensemble de contraintes de précédence est de la forme $t_l + d_l \leq t_{l+1}$, "reliant" t_i à t_j. S' peut être vide.*
- *R' est réduit à un singleton i.e. une seule contrainte de disponibilité, de la forme $t_i \geq r_i$.*
- *L' est réduit à un singleton i.e. une seule contrainte de livraison, de la forme $t_j + d_j \leq l_j$*

Preuve : Soit P un problème d'ordonnancement et $P' =< T', S', R', L' >$ un conflit de P. Considérons deux cas :
- $S' = \emptyset$ il n'y a pas de contrainte de précédence, donc les conflits sont réduits à des tâches isolées. L'inconsistance provient de R' ou de L'. Supposons qu'il existe deux tâches distinctes x_i et x_j inconsistantes. Considérons le sous problème P'' comme P' privé de x_j par exemple. Si P'' reste encore inconsistant à cause de x_i cela contredit la minimalité. Donc il ne peut y avoir qu'une tâche inconsistante dans un conflit P' si $S' = \emptyset$, donc R' et L' sont des singletons.
- $S' \neq \emptyset$ le théorème 1 page 9 implique qu'il existe au moins un chemin inconsistant. Observons un chemin inconsistant partant d'une tâche x_i et arrivant à la tâche x_j en passant par la tâche x_k. Plusieurs cas sont à étudier:
  - Supposons que $R' = \{\{t_i \geq r_i\}, \{t_k \geq r_k\}\}$ et que $L' = \{t_j \leq l_j\}$.
    - Si la suppression de $\{t_i \geq r_i\}$ laisse l'inconsistance, alors la minimalité est contredite (car le sous-problème sans $\{t_i \geq r_i\}$ n'est pas consistant) donc ce n'est pas un conflit, il ne vérifie pas la propriété.
    - Si la suppression de $\{t_i \geq r_i\}$ lève l'inconsistance, alors $\{t_k \geq r_k\}$ est "inutile", on crée le conflit $P'' =< T', S', \{t_i \geq r_i\}, L' >$ qui est bien minimal et qui vérifie bien la propriété.
    - Si la suppression de $\{t_k \geq r_k\}$ laisse l'inconsistance, alors la minimalité est contredite (car le sous-problème sans $\{t_k \geq r_k\}$ n'est pas consistant) donc ce n'est pas un conflit, il ne vérifie pas la propriété.
    - Si la suppression de $\{t_k \geq r_k\}$ lève l'inconsistance, donc $\{t_i \geq r_i\}$ est "inutile", on crée le conflit $P'' =< \{x_k, \ldots, x_j\}, S', \{t_i \geq r_i\}, L' >$ qui est bien minimal et qui vérifie bien la propriété.

En effectuant le même raisonnement on peut montrer que L' est un singleton dans un conflit.
Ceci prouve l'existence et l'unicité des contraintes de disponibilité et de livraison pour un conflit. □

Dans l'exemple 6, on voit que les 3 chemins inconsistants sont minimaux malgré les apparences des notations des chemins.

Exemple 6 *Calcul de conflits.*



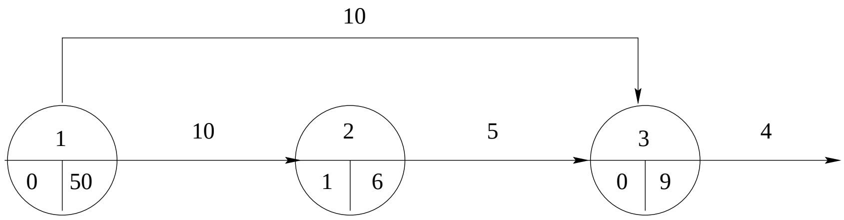
Les conflits sont $(1,2)$,$(2,3)$ *et* $(1,2,3)$. *Interprétons ces résultats.*
- *Pour le conflit* $(1,2)$ *on a :*
  $t_1 \geq r_1$
  $r_1 = 0$
  $t_1 + d_1 \leq t_2$
  $d_1 = 10$
  $t_2 + d_2 \leq l_2$
  $l_2 = 6$

- *Pour le conflit* $(2,3)$ *on a :*
  $t_2 \geq r_2$

$r_2 = 1$
$t_2 + d_2 \leq t_3$
$d_2 = 5$
$t_3 + d_3 \leq l_3$
$l_3 = 9$

– *Pour le conflit* $(1, 2, 3)$ *on a :*
$t_1 \geq r_1$
$r_1 = 0$
$t_1 + d_1 \leq t_2$
$d_1 = 10$
$t_2 + d_2 \leq t_3$
$d_2 = 5$
$t_3 + d_3 \leq l_3$
$l_3 = 9$

Rajoutons la précédence entre la tâche 1 *et la tâche* 3



On obtient un nouveau conflit $(1, 3)$ *:*
$t_1 \geq r_1$
$r_1 = 0$
$d_1 = 10$
$t_1 + d_1 \leq t_3$
$t_3 + d_3 \leq l_3$
$l_3 = 9$

On voit bien que le conflit $(1, 3)$ *n'est pas contenu dans le conflit* $(1, 2, 3)$.

1.3 Construction des conflits

On ne cherche qu'un nombre fini de conflits $nbmaxconflits$.

1.3.1 Propagation arrière

Pour obtenir la liste de tous les conflits, nous utilisons un algorithme de propagation arrière dans le graphe, proposé par [FGT02].

Dans l'algorithme 3 page ci-contre, on calcule d'abord les dates de début au plus tôt de toutes les tâches. Si le problème d'ordonnancement est inconsistant, on cherche alors les tâches x_i telles que $\underline{t}_i + d_i > l_i$ ces tâches constituent le dernier élément d'un chemin inconsistant d'après la caractérisation des conflits de la propriété 1 page précédente. On construit alors les chemins inconsistants en remontant le graphe en arrière. Pour la tâche courante x_j si $r_j > l_i - \sum_{k \in j..i} d_k$, on ajoute r_j et d_j au chemin qui est alors un conflit. Ensuite, on propage à tous les prédécesseurs inconsistants x_l de x_j tel que $\underline{t}_l + \sum_{k \in l..i} d_k > l_i$ après avoir ajouté d_l au chemin courant (on ne propage que sur les éléments qui vont donner des conflits).

Construire tous les conflits dépend, dans le pire des cas, du nombre de chemins du graphe qui est exponentiel (cf [FGT02]). Pour le calcul des dates au plus tôt on utilise l'algorithme 1 page 7.

Algorithme 3: Calcul-Conflit-arr($nbmaxconflits$: int, g : graphe)

% Initialiser les variables globales : l'ensemble des conflits et le nombre de conflits trouvés%
$\mathcal{C} = \{\}$;
$NBCONFLIT = 0$;
début
Calcul des dates au plus tôt;
si *g est consistant* **alors** il n'y a rien a faire;
% S'il reste des conflits à calculer alors on commence la construction du conflit par toutes les tâches qui débutent un conflit %
pour *Toutes les tâches x_i* **faire**
si $plustot[x_i] > l[x_i] - d[x_i]$ *et* $(NBCONFLIT < nbmaxconflits)$ **alors**
Propagation-arr(x_i,$l[x_i] - d[x_i]$,$\{\}$,$nbmaxconflits$);
fin

Procedure `Propagation-arr`(x_i *: int, jalonduree : int, conflit : liste, nbmaxconflits : int*)

début
push($x_i, conflit$);
si $r[x_i] > jalonduree$ **alors**
Ajouter-conflit($conflit, \mathcal{C}$);
$NBCONFLIT = NBCONFLIT + 1$;
pour *toutes les tâches x_j qui précédent x_i* **faire**
si $plustot[x_j] > jalonduree - d[x_j]$*et* $(NBCONFLIT < nbmaxconflits)$ **alors**
Propagation-arr(x_j,$jalonduree - d[x_j]$,$conflit$,$nbmaxconflits$);
fin
% $jalonduree$ représente le jalon du début du conflit moins toutes les durées des tâches du conflit %

1.3.2 Propagation avant

Nous proposons une version de l'algorithme page précédente qui calcule d'abord les dates de début au plus tard puis construit les conflits par propagation en avant. Dans cet algorithme 4 page suivante, on calcule d'abord les dates de début au plus tard de toutes les tâches. Si le problème d'ordonnancement est inconsistant on cherche alors les tâches x_i telles que $\overline{t}_i < r_i$ ces tâches constituent le premier élément d'un chemin inconsistant. On construit alors les chemins inconsistants en avançant dans le graphe. Pour la tâche x_j si $r_i + \sum_{k\in i..j} d_k > l_j$ on ajoute l_j et d_j au chemin qui est alors un conflit. Ensuite on continue avec tous les successeurs inconsistant x_l de x_j tel que $\overline{t}_l < r_i + \sum_{k\in l..j} d_k$ après avoir ajouté d_l au chemin courant (on ne propage que sur les éléments qui vont donner des conflits).

Pour le calcul des dates au plus tard voir l'algorithme 2 page 8. De même, construire tous les conflits dépend dans le pire des cas du nombre de chemins du graphe qui est exponentiel (cf [FGT02]).

1.4 Conclusion

Nous venons de définir la notion de problème d'ordonnancement de tâches et identifier les conflits associés. Dans le chapitre 2, nous allons présenter les travaux basés sur une approche de la résolution de conflits "logiques". Ensuite, les conflits vont être présentés à l'utilisateur, qui va exprimer ces préférences locales sur chaque conflit par une coloration. Grâce à cette coloration nous allons définir des solutions optimales selon les critères que nous établirons. Nous allons alors proposer une aide au chef de projet pour résoudre les conflits trouvés en fonction de ces préférences, grâce à ces critères. Ensuite, dans les chapitres suivants, nous regarderons comment adapter les algorithmes existants, à la résolution des conflits d'un PERT.

Algorithme 4: Calcul-Conflit-av($nbmaxconflits$: int, g : graphe)

% Initialiser les variables globales : l'ensemble des conflits et le nombre de conflits trouvés%
$\mathcal{C} = \{\}$;
$NBCONFLIT = 0$;
début
  Calcul des dates au plus tard;
  si *g est consistant* **alors** Il n'y a rien a faire;
  % S'il reste des conflits à calculer alors on commence la construction du conflit par toutes les tâches qui débutent un conflit %
  pour *toutes les tâches* x_i **faire**
    si $plustard[x_i] < r[x_i]$ *et* $(NBCONFLIT < nbmaxconflits)$ **alors**
      Propagation-av(x_i,$r[x_i] + d[x_i]$,$\{\}$,$nbmaxconflits$);
fin

Procedure `Propagation-av(`x_i *: int, dispoduree : int, conflit : liste, nbmaxconflits: int*`)`

début
  Ajouter-tâche-conflit($x_i, conflit$);
  si $l[x_i] < dispoduree$ **alors**
    Ajouter-conflit($conflit, \mathcal{C}$);
    $NBCONFLIT = NBCONFLIT + 1$;
  pour *toutes les tâches* x_j *qui succèdent à* x_i **faire**
    si $plustard[x_j] < dispoduree + d[x_j]$*et* $(NBCONFLIT < nbmaxconflits)$ **alors**
      Propagation-av(x_j,$dispoduree + d[x_j]$,$conflit$,$nbmaxconflits$);
fin

% *dispoduree* représente la date de disponibilité du début du conflit plus toutes les durées des tâches du conflit %

Chapitre 2

Formalisation de la résolution de conflits à l'aide de préférences

Dans la partie précédente, nous avons vu comment calculer les conflits dans un PERT. Nous nous intéressons maintenant à la résolution de ces conflits d'un point de vue purement "logique" *i.e.* en considérant chaque élément d'un conflit comme une formule logique. Nous nous intéresserons ensuite à trouver une solution optimale au problème de conflits pour des critères que nous définirons.

2.1 Définition du problème dans un contexte logique

En logique, il existe également la notion de conflit et de solution.

Définition 10 (Conflit) *Un ensemble de formules $F = \{f_i, i = 1..n\}$ est appelé conflit si et seulement s'il satisfait les deux conditions suivantes :*
(i) l'ensemble des formules de F est classiquement incohérent i.e. $F \vdash \perp$
(ii) l'ensemble des formules de F est minimal parmi ceux vérifiant (i), i.e. $\forall f_j \in F, F \setminus \{f_j\} \not\vdash \perp$

Notation : Un ensemble $\mathcal{C} = \{c_i, i = 1..n\}$ de conflits est appelé problème de conflits.

Remarquons qu'un ensemble S, qui contient au moins une formule de chaque conflit, permet de résoudre tous les conflits, car les conflits sont minimaux pour l'inclusion d'ensembles.

Définition 11 (Solution) *Soit $\mathcal{C} = \{c_i, i = 1..n\}$ un problème de conflits. S est une solution de $\mathcal{C}$ si et seulement si S est un sous-ensemble de $N = \cup_i c_i$ tel que $\forall i, c_i \cap S \neq \emptyset$.*

Exemple 7 *Soit l'ensemble des conflits $\mathcal{C} = \{\{a, b\}, \{b, c\}\}$, nous obtenons les solutions suivantes :*
$S_1 = \{a, b, c\}$
$S_2 = \{a, b\}$
$S_3 = \{b, c\}$
$S_4 = \{a, c\}$
$S_5 = \{b\}$

Lorsque le problème d'ordonnancement est inconsistant, les algorithmes du chapitre précédent permettent d'obtenir les ensembles de conflits minimaux (au sens de l'inclusion). En considérant les inéquations et les équations comme des formules "logiques" et en utilisant la caractérisation d'un conflit du théorème 1 page 11, nous identifions les solutions d'un conflit en logique (définition 11) et les solutions d'un conflit dans un problème d'ordonnancement (définition 9 page 11). Ainsi une solution est un sous-ensemble S contenant au moins une formule de chaque conflit.

2.2 Préférences

Il nous paraît souhaitable de ne pas choisir aléatoirement une solution pour résoudre un conflit. L'idée de [CLSM00] est de demander à l'utilisateur quelles sont ses **préférences**, et ce **par conflit**, afin de lui proposer une solution adaptée. Ici on va étendre cette notion à 3-strates.

2.2.1 Préférences à 3 strates

Nous demandons à l'utilisateur de classer les formules d'un conflit, selon ses préférences. Il nous indique pour chaque formule d'un conflit donné s'il lui est impossible de la changer ou bien s'il peut éventuellement la supprimer ou bien s'il lui est très facile de la supprimer. Nous obtenons donc 3 classes :

- Les non-modifiables (rouge)
- Les éventuellement modifiables (bleu)
- Les modifiables (vert)

Ainsi nous espérons pouvoir calculer des solutions qui satisferont au mieux les choix de l'utilisateur.

Nous introduisons la notion de 3-strates à l'intérieur d'un conflit.

Définition 12 (Conflit 3-stratifié) *Soit $\mathcal{C} = \{c_i, i = 1..n\}$ un problème de conflits. Un conflit sera dit 3-stratifié s'il est de la forme :*
$c_i = r(c_i) \cup b(c_i) \cup v(c_i)$ *avec :*

- $r(c_i) = \{$*formules non-modifiables de* $c_i\}$.
- $b(c_i) = \{$*formules éventuellement modifiables de* $c_i\}$.
- $v(c_i) = \{$*formules modifiables de* $c_i\}$.
  où $r(c_i), b(c_i), v(c_i)$ *forment une partition de* c_i.

Remarque 7 *Les $r(c_i)$ ou les $b(c_i)$ ou les $v(c_i)$ peuvent éventuellement être vides. Par contre* $r(c_i) \cup b(c_i) \cup v(c_i) \neq \emptyset$

Notation : Dans les exemples nous représenterons dans les conflits

- les formules non-modifiables (les $r(c_i)$) en **gras**.
- les formules éventuellement modifiables (les $b(c_i)$) écrites normalement.
- les formules modifiables (les $v(c_i)$) <u>soulignées</u>.

La définition s'étend à un problème de conflits.

Définition 13 (Problème de conflits 3-stratifié) *Soit $\mathcal{C} = \{c_i, i = 1..n\}$ un problème de conflits est 3-stratifié si tous ses conflits sont 3-stratifiés.*

Définition 14 (R V B) *Soit $\mathcal{C} = \{c_i, i = 1..n\}$ un problème de conflits 3-stratifié on pose :*

- $\mathcal{R} = \cup_{j\in\{1..n\}} r(c_j)$ *ensemble des formules non-modifiables de* $\mathcal{C}$[1].
- $B = \cup_{j\in\{1..n\}} b(c_j)$ *ensemble des formules éventuellement modifiables de* $\mathcal{C}$.
- $V = \cup_{j\in\{1..n\}} v(c_j)$ *ensemble des formules modifiables de* $\mathcal{C}$.

Remarque 8 *$\mathcal{R}, B, V$ ne sont pas forcément disjoints deux à deux, puisqu'une même formule peut être rouge dans un conflit et verte dans un autre.*

Exemple 8 $\mathcal{C} = \{\{\mathbf{a_1}, \underline{a_2}, a_3\}, \{\mathbf{a_4}, a_5, \underline{a_6}\}, \{\mathbf{a_1}, a_4, \underline{a_2}\}\}$

Les choix de l'utilisateur peuvent être "irréalistes" si un conflit ne contient que des formules non-modifiables. Le problème est alors incohérent.

Définition 15 (Problème incohérent) *Soit $\mathcal{C} = \{c_i, i = 1..n\}$ un problème de conflits 3-stratifié. $\mathcal{C}$ est incohérent si $\exists i$ tel que $c_i \setminus \mathcal{R} = \emptyset$, i.e. $c_i \subseteq \mathcal{R}$. Sinon il est dit cohérent.*

2.2.2 Passage d'un problème de conflits à 3-strates à un problème de conflits à 2-strates

Comme il est interdit d'agir sur des formules non-modifiables, nous devons travailler sur les conflits privés de ces formules. Si un problème n'est pas incohérent, nous passons d'un problème 3-stratifié à un nouveau problème 2-stratifié "équivalent", par un procédé de nettoyage (cf algorithme 5 page suivante). Pour cela, nous ôtons de tous les conflits les éléments de $\mathcal{R}$ pour obtenir un nouveau **problème propre** avec uniquement 2 strates, mais non minimal. Ensuite, nous ôtons les ensembles de formules non minimaux pour l'inclusion pour obtenir un problème 2-stratifié.

Remarque 9 *Un problème propre est cohérent si et seulement s'il ne contient pas de conflit $\{\}$.*

Algorithme 5: Algorithme de passage d'un problème 3-stratifié à un problème 2-stratifié en éliminant les éléments non-modifiables

$\mathcal{C} = \{c_i, i = 1, \ldots, n\}$
début
  Calculer $\mathcal{R} = \cup_i r(c_i)$;
  % On supprime les formules non-modifiables %
  pour *i de 1 à n* **faire**
    pour $x \in c_i$ **faire**
      si $x \in \mathcal{R}$ **alors** $c_i = c_i - x$;
  % On minimise les ensembles de formules restants dans le problème propre obtenu %
  nettoyage-pas-fini := VRAI;
  tant que *nettoyage-pas-fini* **faire**
    si $\exists c_i, c_j \in \mathcal{C}$ *tels que* $c_i \subseteq c_j$ **alors**
      $\mathcal{C} = \mathcal{C} \setminus c_j$;
    sinon
      nettoyage-pas-fini := FAUX ;
fin

Remarque 10 *Dans l'exemple 8 page ci-contre après suppression des formules non-modifiables, on obtient le problème propre suivant :* $\mathcal{C}^{'} = \{\{\underline{a_2}, a_3\}, \{a_5, \underline{a_6}\}, \{\underline{a_2}\}\}$.
On s'aperçoit que $\mathcal{C}^{'}$ *contient* $\{\underline{a_2}\}$ *qui est contenu dans* $\{\underline{a_2}, a_3\}$.
On ne garde que les ensembles de formules minimaux au sens de l'inclusion, on a alors un nouveau problème de conflit 2-stratifié $\mathcal{C}^{''} = \{\{a_5, \underline{a_6}\}, \{\underline{a_2}\}\}$.

Propriété 2 *Soit* $\mathcal{C} = \{c_i, i = 1..n\}$ *un problème de conflits,* $\mathcal{C}' = \{c_i^{'}, i = 1..n\}$ *l'ensemble de conflit 2-stratifiés associé au problème propre issu de* $\mathcal{C}$*, noté* $\mathcal{C}_p$*. Toute solution de* $\mathcal{C}'$ *est solution de* $\mathcal{C}$*.*

Preuve : Soit $\mathcal{C} = \{c_i, i = 1..n\}$ un problème de conflits, $\mathcal{C}' = \{c_i^{'}, i = 1..n\}$ l'ensemble de conflit 2-stratifiés associé au problème propre issu de $\mathcal{C}$, noté $\mathcal{C}_p$. Soit S une solution de $\mathcal{C}'$ *i.e.* $\forall i, c_i^{'} \cap S \neq \emptyset$. Comme $\forall c_i \in \mathcal{C}_p, \exists c_j \in \mathcal{C}', c_j \subset c_i$ on a $\forall i, c_i \cap S \neq \emptyset$ donc S solution de $\mathcal{C}$. □

Dans la suite nous ne travaillons que sur des problèmes de conflits à 2-strates. Ce choix ne provoque aucune perte d'information d'après la propriété 2.

2.3 Différents critères de résolution d'un problème de conflits

Notation : S_1 et S_2 dénotant des solutions d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$ et $\leq_x$ une relation d'ordre alors $S_1 \leq_x S_2$ signifie que S_2 est meilleure que S_1 selon "x".

Nous étudions plusieurs critères permettant de comparer les solutions pour prendre en compte les préférences de l'utilisateur.
Nous étudions les critères sur des problèmes de conflits 2-stratifiés.

2.3.1 Critères classiques sans préférence

Dans [CLSM00] sont proposés plusieurs critères que nous reprenons ici. Certains de ces critères sont dits sans préférence, car ils ne tiennent pas compte des préférences de l'utilisateur.

2.3.1.1 Solution minimale (au sens de l'inclusion)

Définition 16 (Solution minimale) *Soit* $\mathcal{C} = \{c_i, i = 1..n\}$ *un problème de conflits, soit* S_1, S_2 *deux solutions de* $\mathcal{C}$*.* $S_1 \geq_{incl} S_2$ *si et seulement si* $S_1 \subseteq S_2$*. Les solutions optimales pour cette relation* $\geq_{incl}$ *sont*

[1]Attention à ne pas confondre avec R : l'ensemble des contraintes de disponibilité

appelées solutions minimales.

Propriété 3 *Soit* $\mathcal{C} = \{c_i, i = 1..n\}$ *un problème de conflits.* S *est une solution minimale de* $\mathcal{C}$ *si et seulement si* S *est un sous-ensemble de* $N = \cup_i c_i$ *vérifiant la propriété (I) suivante :*
- $\forall i, c_i \cap S \neq \emptyset$
- $\forall x \in S, \exists i$ *tel que* $c_i \cap S = \{x\}$

Preuve : ($\Rightarrow$) Soit S une solution minimale de $\mathcal{C}$. Supposons qu'elle ne vérifie pas la propriété (I) :
- Soit $\exists i, c_i \cap S = \emptyset$ (impossible car S solution *i.e.* $\forall i, c_i \cap S \neq \emptyset$)
- Soit $\exists x \in S, \forall c_i, c_i \cap S \neq \{x\}$ donc, $\forall c_j, \exists y_j \in S, y_j \neq x$ tel que $\{x, y_j\} \subseteq c_j \cap S$. On a alors que $S' = S \setminus \{x\}$ est une solution et $S' \subset S$ ce qui contredit le fait que S soit minimale.

($\Leftarrow$) Soit S un sous-ensemble de $N = \cup_i c_i$ qui vérifie la propriété (I). Supposons qu'elle n'est pas minimale *i.e.* il existe S' solution telle que $S' \subsetneq S$. Comme S et S' sont des solutions on a $\forall i, S' \cap c_i \neq \emptyset$ et $\forall i S \cap c_i \neq \emptyset$. Soit $y \in S \setminus S'$ (y existe car $S' \subsetneq S$). On considère le conflit c_i tel que $\{y\} = S \cap c_i$ (existe car S vérifie la propriété (I)). Or $\exists x \in S'$ tel que $x \in S' \cap c_i$ (car S' solution donc $\forall j, c_j \cap S' \neq \emptyset$) et $x \neq y$ donc $\{x, y\} \subseteq S \cap c_i$ ce qui contredit la propriété (I); donc l'hypothèse S n'est pas minimale est fausse *i.e.* S est minimale. □

Une solution minimale S résout tous les conflits, mais en plus chaque formule de S permet de résoudre à elle seule au moins un conflit.

Notation : les solutions minimales sur les exemples sont soulignées.
Dans l'exemple 7 page 15.
$\mathcal{C} = \{\{a, b\}, \{b, c\}\}$
$S_1 = \{a, b, c\}$
$S_2 = \{a, b\}$
$S_3 = \{b, c\}$
$\underline{S_4} = \{a, c\}$
$\underline{S_5} = \{b\}$
S_4 et S_5 sont les deux seules solutions minimales.

2.3.1.2 Solution parcimonieuse

La solution parcimonieuse contient le moins possible d'éléments c'est donc une solution minimale au sens de la cardinalité.

Définition 17 (Solution parcimonieuse) *Soit* S_1, S_2 *deux solutions d'un problème de conflits* $\mathcal{C} = \{c_i, i = 1..n\}$. $S_1 \geq_{card} S_2$ *si et seulement si* $|S_1| \leq |S_2|$. *La solution optimale pour cette relation d'ordre est appelée solution parcimonieuse.*

Théorème 2 *Dans un problème de conflits* $\mathcal{C}$, *toute solution respectant la parcimonie (minimalité selon la cardinalité) respecte la minimalité selon l'inclusion ensembliste, i.e. toute solution parcimonieuse est minimale pour l'inclusion.*

Preuve : Soit S une solution parcimonieuse pour un problème de conflits $\mathcal{C}$, *i.e.* $|S|$ est minimal, donc quelle que soit S' une autre solution telle que $S \neq S'$ et $|S| \leq |S'|$ on a $S' \not\subseteq S$; donc S est minimale au sens de l'inclusion ensembliste. □

2.3.2 Nouveaux critères sans préférence

En nous inspirant de la minimalité (définition 16 page précédente) nous voulons que chaque conflit ne contienne qu'un seul élément de la solution.

2.3.2.1 Solution minimale forte

Définition 18 (Solution minimale forte) *Soit* $\mathcal{C} = \{c_i, i = 1..n\}$ *un problème de conflits.* S *est une solution minimale forte de* $\mathcal{C}$ *si et seulement si* S *est un sous-ensemble de* $N = \cup_i c_i$ *tel que :*
- $\forall i, c_i \cap S \neq \emptyset$
- $\forall i, \exists x \in S$ *tel que* $c_i \cap S = \{x\}$

Cela revient à n'enlever qu'un seul élément par conflit.

Remarque 11 *Il se peut qu'il n'existe pas de solution minimale forte, cf exemple 9.*

Exemple 9 $\mathcal{C} = \{\{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}\}$ *Il n'y a pas de solution minimale forte.*
$\underline{S_1} = \{a_1, a_2\}$
$\underline{S_2} = \{a_1, a_3\}$
$\underline{S_3} = \{a_2, a_3\}$

Propriété 4 *Soit $\mathcal{C} = \{c_i, i = 1..n\}$ un problème de conflits, toute solution minimale forte est minimale.*

Preuve : Soit S une solution minimale forte d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$. $S \subseteq \cup_i c_i$, donc $\forall x \in S, \exists i$, tel que $x \in c_i$. Or $c_i \cap S$ réduit à un singleton donc $c_i \cap S = \{x\}$ $\square$

Remarque 12 *Une solution minimale forte lorsqu'elle existe, n'est pas forcément unique. Prenons par exemple l'ensemble de conflits $\mathcal{C} = \{\{a_1, a_2\}, \{a_3\}\}$. Il possède deux solutions minimales fortes $S_1 = \{a_1, a_3\}$ et $S_2 = \{a_2, a_3\}$.*

Propriété 5 *Soit $\mathcal{C} = \{c_i, i = 1..n\}$ un problème de conflits, S est une solution minimale et $|S| = |\mathcal{C}|$ alors S est solution minimale forte.*

Preuve : Soit $\mathcal{C} = \{c_i, i = 1..n\}$ un problème de conflits. S est une solution minimale de $\mathcal{C}$ on a :
- $\forall i, c_i \cap S \neq \emptyset$
- $\forall x \in S, \exists i$ tel que $c_i \cap S = \{x\}$

Considérons l'application $f : S \rightarrow \mathcal{C}$ qui à chaque élément $x \in S$ associe c_x un élément de $\mathcal{C}$ tel que $c_x \cap S = \{x\}$ (ce c_x existe car S est minimale).

La fonction f est injective.
Preuve : Prenons $x_1 \in S$ et $x_2 \in S$ tel que $x_1 \neq x_2$ alors $f(x_1) = c_{x_1}$ et $f(x_2) = c_{x_2}$. Supposons que $f(x_1) = f(x_2)$ et cherchons une contradiction.
$f(x_1) = f(x_2)$ implique que $c_{x_1} = c_{x_2}$. Or $c_{x_1} \cap S = \{x_1\}$ et $c_{x_2} \cap S = \{x_2\}$ donc comme $c_{x_1} = c_{x_2}$ on doit avoir $x_1 = x_2$. Contradiction donc $f(x_1) \neq f(x_2)$ donc f est injective. $\square$

De plus S et $\mathcal{C}$ sont des ensembles finis de même cardinalité ($|S| = |\mathcal{C}|$), f étant injective donc f est surjective, on a donc que f est une bijection entre l'ensemble des conflits et les éléments de S.
A chaque conflit $c_i \in \mathcal{C}$ on peut associer un $x \in S$ c'est à dire :
- $\forall i, c_i \cap S \neq \emptyset$
- $\forall i, \exists x \in S$ tel que $c_i \cap S = \{x\}$

Donc S est une solution minimale forte.
$\square$

Remarque 13 *Une solution minimale forte, lorsqu'elle existe, n'est pas forcément parcimonieuse. Dans l'exemple 18 page 26, la solution S_1 est parcimonieuse et non-minimale forte et, la solution S_4 est minimale forte et non parcimonieuse.*

2.3.2.2 Solution Pmaxmin

Présence max Nous voulons maintenant totaliser le nombre d'apparitions d'un élément de S dans chaque conflit.

Définition 19 (Pmax) *Soit $\mathcal{C} = \{c_i, i = 1..n\}$ un problème de conflits, nous définissons le nombre d'apparitions des éléments d'une solution S dans chaque conflit, noté Pmax.*

$$Pmax(S) = \sum_i |S \cap c_i|$$

Exemple 10 $\mathcal{C} = \{\{a, c\}, \{a, d\}, \{a, b, e\}, \{a, b, f\}, \{b, g\}, \{b, h\}\}$

$\underline{S_1} = \{a, b\}$
$\underline{S_2} = \{a, g, h\}$
$\underline{S_3} = \{b, d, c\}$
$\underline{S_4} = \{c, d, e, f, g, h\}$

Exemple 10 page précédente	S_1	S_2	S_3	S_4
Pmax (def 19 page précédente)	8	6	6	6
Cardinalité (def 17 page 18)	2	3	3	6

Propriété 6 (Caractérisation d'une solution minimale forte) *Une solution S vérifiant $Pmax(S) = |S|$ est minimale forte .*

Preuve : Soit S une solution *i.e.* $\forall i, c_i \cap S \neq \emptyset$ telle que $Pmax(S) = |S|$ *i.e.*

$$\sum_i |S \cap c_i| = |S| \Rightarrow \forall i |S \cap c_i| = 1 \text{ car } \forall i, c_i \cap S \neq \emptyset$$

donc $\forall i, \exists x \in S$ tel que $c_i \cap S = \{x\}$ donc S est une solution minimale forte. $\square$

Présence maxmin Nous minimisons Pmax.

Définition 20 (Pmaxmin) *Soit S_1, S_2 deux solutions d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$.*
$S_1 \leq_{Pmaxmin} S_2$ si et seulement si $Pmax(S_1) \geq Pmax(S_2)$

La solution la meilleure au sens de Pmaxmin correspond à une sorte d'économie : on enlève les formules qui apparaissent le moins sur l'ensemble des conflits.

Remarque 14 *L'exemple 10 page précédente montre que la solution S_1 est parcimonieuse, mais n'est pas optimale selon la relation $\leq_{Pmaxmin}$ et réciproquement une solution optimale selon la relation $\leq_{Pmaxmin}$ n'est pas forcément parcimonieuse.*

Remarque 15 *Dans l'exemple 9 page précédente il n'y a pas de solution minimale forte et toutes les solutions sont optimales pour la relation $\leq_{Pmaxmin}$. Donc une solution optimale selon la relation $\leq_{Pmaxmin}$ n'est pas forcément minimale forte.*

Théorème 3 (Minimalité au sens de l'inclusion de Pmaxmin) *Soit S_1 et S_2 deux solutions d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$ telles que $S_1 \geq_{incl} S_2$ alors $S_1 \geq_{Pmaxmin} S_2$*

Preuve : $S_1 \geq_{incl} S_2 \Rightarrow S_1 \subseteq S_2 \Rightarrow \forall i, S_1 \cap c_i \subseteq S_2 \cap c_i$
$\Rightarrow \forall i, |S_1 \cap c_i| \leq |S_2 \cap c_i|$
$\Rightarrow \sum_i |S_1 \cap c_i| \leq \sum_i |S_2 \cap c_i|$
$\Rightarrow S_1 \geq_{Pmaxmin} S_2$ $\square$

Remarque 16 *De la même manière on démontre le théorème avec des inégalités strictes, donc $\geq_{Pmaxmin}$ est un raffinement de $\geq_{incl}$. De plus contrairement à $\geq_{incl}$, $\geq_{Pmaxmin}$ est un pré-ordre complet.*

Corollaire 1 *La solution optimale pour Pmaxmin est minimale.*

Preuve : Supposons que S_1 la solution optimale pour Pmaxmin ne soit pas minimale. Il existerait alors une solution S_0 telle que $S_0 \subsetneq S_1$ donc $S_0 <_{Pmaxmin} S_1$ ce qui est impossible car S_1 est optimale pour Pmaxmin. $\square$

Remarque 17 *Dans l'exemple 10 page précédente S_1 n'est pas optimale pour Pmaxmin mais elle est minimale.*

2.3.3 Critères classiques avec préférence

Nous allons prendre en compte les préférences de l'utilisateur et définir des critères pour exhiber les solutions associés. Dans [CLSM00] sont proposés plusieurs critères avec préférences que nous reprenons ici.

2.3.3.1 Principe de préférence

Une solution sera dite préférée si tous ses éléments rencontrent *au moins* une fois l'ensemble des formules modifiables d'un conflit.

Définition 21 (Solution préférée) *Soit $\mathcal{C} = \{c_i, i = 1..n\}$ un problème de conflits. Une solution S est dite préférée si et seulement si S est une solution telle que : $\forall x \in S, \exists j$ tel que $x \in v(c_j)$[2].*

[2] *i.e.* $S \subseteq V = \cup_j v(c_j)$

Définition 22 (Solution minimale préférée) *Soit $\mathcal{C} = \{c_i, i = 1..n\}$ un problème de conflits. S est une solution minimale préférée si et seulement si S est une solution minimale (définition 16 page 17) et préférée (définition 21 page précédente)*

Notation : Les étoiles ★ représentent les solutions minimales préférées.

Remarque 18 *Les formules d'un conflit peuvent être toutes "bleues", il existe alors des exemples où il n'y a pas de solution minimale préférée, cf exemple 11.*

Exemple 11 $\mathcal{C} = \{\{a_1, \underline{a_2}\}, \{a_1, a_3\}, \{a_3, \underline{a_2}\}\}$ *Ce problème de conflits ne possède pas de solution minimale préférée.*
$\underline{S_1} = \{a_1, a_2\}$
$\underline{S_2} = \{a_1, a_3\}$
$\underline{S_3} = \{a_2, a_3\}$

Propriété 7 *Si dans un problème de conflits 2-stratifié, chaque conflit possède deux strates non-vides alors il existe toujours une solution minimale préférée.*

Preuve : Par récurrence sur le nombre de conflits du problème.
- Pour $n = 1$, nous avons un seul conflit c_1 à 2-strates. Il existe donc une solution minimale préférée (toutes celles constituées d'un élément de $v(c_1)$).
- Supposons que pour un problème 2-stratifié à n conflits il existe une solution S minimale préférée. Vérifions alors que pour $n + 1$ conflits il existe une solution minimale préférée. On rajoute un conflit c_{n+1} qui est 2-stratifié. S'il existe un élément de c_{n+1} qui appartient à S, alors S est une solution minimale preférée pour le problème de conflits 2-stratifiés à $n + 1$ conflits. Sinon, comme c_{n+1} est 2-stratifié et qu'aucun élément de c_{n+1} n'appartient à S, il suffit de rajouter un x de $v(c_{n+1})$ à S pour obtenir une solution minimale preférée du problème de conflits 2-stratifiés à $n + 1$ conflits.

Nous avons bien montré que si chaque conflit possède 2-strates non-vides alors il existe toujours une solution minimale preférée. □

2.3.3.2 Solution éli-Préférée

Nous introduisons la préférence dite élitiste d'après [CLSM00]

Définition 23 (Eli-Préférence) *Soit S_1, S_2 deux solutions minimales préférées d'un problème de conflits $\mathcal{C} = \{i = 1..n, c_i\}$. S_2 est éli-Préférée à S_1 (noté $S_1 \leq_{eli} S_2$) si et seulement si $\forall x \in S_2 \setminus S_1, \exists y \in S_1 \setminus S_2$ et $\exists c_i \in \mathcal{C}$ tel que $y \in c_i \setminus v(c_i)$ et $x \in v(c_i)$.*

Exemple 12 $\mathcal{C} = \{\{a_1, \underline{a_2, a_3}\}, \{a_2, a_4, \underline{a_1}\}, \{a_2, a_4, \underline{a_3}\}\}$
$\underline{S_1} = \{a_2\}$★
$\underline{S_2} = \{a_1, a_3\}$★
$\underline{S_3} = \{a_1, a_4\}$
$\underline{S_4} = \{a_3, a_4\}$

Nous ne pouvons donc pas trancher entre $S_1 \leq_{eli} S_2$ et $S_2 \leq_{eli} S_1$; Nous remarquons que, si $x \in S_2 \setminus S_1$ est meilleur que $y \in S_1 \setminus S_2$ dans un conflit, il peut y avoir un $z \in S_1 \setminus S_2$ aussi bon que x dans le même conflit. (par exemple pour le conflit $\{a_1, a_2, a_3\}$ de S_1, a_2 de S_1 résout mieux le conflit que a_1 de S_2, mais il existe aussi a_3 de S_2 qui résout aussi bien le conflit que a_2 de S_1).

Rappel : Soit A et B deux éléments et $\leq$ une relation entre éléments, on définit :
$A < B \Leftrightarrow A \leq B$ et $\neg(B \leq A)$
$A \equiv B \Leftrightarrow A \leq B$ et $B \leq A$
$A \cong B \Leftrightarrow \neg(A \leq B)$ et $\neg(B \leq A)$

Rappel : La relation $\leq$ sera quasi-transitive si la relation $<$ associée à la relation $\leq$ est transitive.
Rappel : La relation $\leq$ sera complète si tous les éléments sont comparables.

Remarque 19 *Toute relation transitive est évidement quasi-transitive.*

L'exemple qui suit est un contre-exemple qui montre que l'éli-préférence n'est pas une relation quasi-transitive.

Exemple 13 $\mathcal{C} = \{c_1 = \{x, c, \underline{b, d}\}, c_2 = \{z, c, a, \underline{e}\}, c_3 = \{r, e, c, \underline{b}\}, c_4 = \{u, a, \underline{e, c}\}, c_5 = \{w, e, \underline{c, a}\}\}$

$\underline{S_1} = \{c\}$ ★
$\underline{S_2} = \{a, b\}$ ★
$\underline{S_3} = \{d, e\}$ ★
$\underline{S_4} = \{e, b\}$ ★
$\underline{S_5} = \{x, e\}$
$\underline{S_6} = \{r, x, a\}$
$\underline{S_7} = \{r, d, a\}$
$\underline{S_8} = \{w, u, z, b\}$
$\underline{S_9} = \{w, u, r, z, d\}$
$\underline{S_{10}} = \{w, u, r, z, x\}$

Nous établissons notre contre-exemple grâce à S_1, S_2 et S_3. Remarquons que les trois solutions sont disjointes deux à deux.

Montrons que

- $S_2 \leq_{eli} S_1$
  $\forall x \in S_1 = \{c\}, \exists y \in S_2 = \{a, b\}, \exists c_4 = \{u, a, \underline{e, c}\}$

- $S_1 \leq_{eli} S_3$
  $\forall x \in S_3 = \{d, e\}, \exists y \in S_2 = \{c\}$,
  Pour $d, \exists c_1 = \{x, c, \underline{b, d}\}$
  Pour $e, \exists c_2 = \{z, c, \underline{a, e}\}$

- $S_3 \leq_{eli} S_2$
  $\forall x \in S_2 = \{a, b\}, \exists y \in S_3 = \{d, e\}$,
  Pour $a, \exists c_5 = \{w, e, \underline{c, a}\}$
  Pour $b, \exists c_3 = \{r, e, \underline{c, b}\}$

- $S_3 \leq_{eli} S_1$
  $\forall x \in S_1 = \{c\}, \exists y \in S_3 = \{d, e\}, \exists c_5 = \{w, e, \underline{c, a}\}$

- $S_1 \not\leq_{eli} S_2$
  Pour $a \in S_2, c_5 = \{w, e, \underline{c, a}\}$ *seul conflit où a apparaît comme préféré et c n'apparaît pas comme non-préféré.*

- $S_2 \not\leq_{eli} S_3$
  Pour $d \in S_3, c_1 = \{x, c, \underline{b, d}\}$ *seul conflit où d apparaît comme préféré et ni a ni b n'apparaissent pas comme non-préféré.*

En résumé nous avons :

- $S_1 \leq_{eli} S_3$ *et* $S_3 \leq_{eli} S_1$ *donc* $S_1 \equiv S_3$
- $S_3 \leq_{eli} S_2$ *et* $S_2 \not\leq_{eli} S_3$ *donc* $S_3 <_{eli} S_2$
- $S_2 \leq_{eli} S_1$ *et* $S_1 \not\leq_{eli} S_2$ *donc* $S_2 <_{eli} S_1$

Donc $\leq_{eli}$ n'est pas quasi-transitive.

2.3.4 Nouveaux critères avec préférence

2.3.4.1 Adaptation des critères classiques

Nous introduisons alors éli-Préférence forte, qui tient compte du problème évoqué à l'exemple 12 page précédente.

Solution éli-Préférence forte

Définition 24 (Eli-Préférence forte) *Soit S_1, S_2 deux solutions minimales préférées d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$. $S_1 \leq_{eli-F} S_2$ si et seulement si $\forall x \in S_2 \setminus S_1, \exists c_i \in \mathcal{C}$ tel que*

- $x \in v(c_i)$.
- $\exists y \in S_1 \setminus S_2, y \in c_i \setminus v(c_i)$

– $\nexists z \in S_1 \setminus S_2, z \in v(c_i)$

Donc l'éli-préférence forte nous fera choisir S_2 alors que l'éli-préférence ne pouvait départager S_1 et S_2. Nous imposons que, quand $x \in S_2 \setminus S_1$ est meilleur que $y \in S_1 \setminus S_2$ dans un conflit alors il ne peut pas y avoir un $z \in S_1 \setminus S_2$ aussi bon que x.

Revenons sur l'exemple 12 page 21 on a :
- $S_1 \leq_{eli} S_2$ et $S_2 \leq_{eli} S_1$
- $S_1 \leq_{eli-F} S_2$ et $S_2 \nleq_{eli-F} S_1$

Par contre l'exemple 13 page 21 montre de la même manière que l'éli-préférence forte n'est pas non plus une relation quasi-transitive.

Propriété 8 *Soit S une solution éli-Préférée forte de $\mathcal{C}$ alors elle est éli-Préférée.*

Preuve : immédiat d'après la définition 24 page ci-contre □

Solution "parfaite"

Nous définissons la notion de solution parfaite *i.e.* solution qui ne contient que des formules toujours modifiables (vertes) *dans tous* les conflits.

Définition 25 (Solution parfaite) *Une solution minimale S, d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$, telle que $\forall x \in S, \nexists i$ tel que $x \in c_i \setminus v(c_i)$ sera dite parfaite*[3].

Dans l'exemple 14, S_6 est parfaite.

Exemple 14 $\mathcal{C} = \{\{a_6, \underline{a_1, a_2}\}, \{a_7, \underline{a_2, a_3}\}, \{a_3, \underline{a_4, a_8}\}, \{a_4, \underline{a_5}\}\}$. *Nous ne donnons que les solutions minimales.*

$\underline{S_1} = \{a_2, a_4\}$ ★
$\underline{S_2} = \{a_1, a_3, a_4\}$ ★
$\underline{S_3} = \{a_1, a_3, a_5\}$ ★
$\underline{S_4} = \{a_1, a_4, a_7\}$
$\underline{S_5} = \{a_2, a_3, a_5\}$ ★
$\underline{S_6} = \{a_2, a_5, a_8\}$ ★ *solution parfaite*
$\underline{S_7} = \{a_3, a_4, a_6\}$
$\underline{S_8} = \{a_3, a_5, a_6\}$
$\underline{S_9} = \{a_4, a_6, a_7\}$
$\underline{S_{10}} = \{a_1, a_5, a_8, a_7\}$
$\underline{S_{11}} = \{a_5, a_6, a_8, a_7\}$

Remarque 20 *Si $B \cap V = \emptyset$ et les conflits sont tous 2-stratifiés, alors il existe une solution parfaite contenue dans V. Par contre la réciproque n'est pas vraie comme le montre l'exemple 14.*

Définition 26 (Problème parfait) *Un problème de conflits 2-stratifié est dit parfait s'il existe une solution parfaite. Sinon il est imparfait.*

Dans l'exemple suivant, nous avons repris les conflits de l'exemple 7 page 15. Avec les préférences de l'utilisateur suivantes, le problème est imparfait.

Exemple 15 $\mathcal{C} = \{\{a, \underline{b}\}, \{b, \underline{c}\}\}$ *Problème de conflits imparfait.*

$S_1 = \{a, b, c\}$
$S_2 = \{a, b\}$
$S_3 = \{b, c\}$
$\underline{S_4} = \{a, c\}$
$\underline{S_5} = \{b\}$

Solution minimale préférée forte (Pref-F)

Une autre approche consiste à exiger que, dans tous les conflits, toutes les formules de la solution choisie ne rencontrent que l'ensemble des formules modifiables.

Définition 27 (Solution minimale préférée forte) *Soit $\mathcal{C} = \{c_i, i = 1..n\}$ un problème de conflits. S est une solution minimale préférée forte de $\mathcal{C}$ si et seulement si S est une solution minimale préférée selon la définition 22 page 21* [4] *telle que :*

[3] *i.e.* $\forall x \in S, \forall i, x \in v(c_i)$

[4] $\forall i, c_i \cap S \neq \emptyset$ (solution), $\forall x \in S, \exists i$ tel que $c_i \cap S = \{x\}$ (minimale), et $\forall x \in S, \exists j$ tel que $x \in v(c_j)$ (préférée)

$\forall x \in S, \forall i$
- *soit* $c_i \cap \{x\} = \emptyset$ *i.e.* $x \notin c_i$
- *soit* $x \in v(c_i)$

Propriété 9 *Si S est une solution minimale préférée forte de $\mathcal{C}$ alors elle est minimale préférée.*

Preuve : Les solutions minimales préférées fortes sont choisies parmi les solutions minimales préférées. □

Théorème 4 *Soit une solution minimale S d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$. S est préférée forte $\Leftrightarrow$ S est parfaite.*

Preuve : ($\Rightarrow$) Soit S une solution minimale préférée forte d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$, raisonnons par l'absurde et supposons qu'elle n'est pas parfaite *i.e.* $\exists x' \in S, \exists i$ tel que $x' \in c_i \setminus v(c_i)$ or on sait que $\forall x \in S$, donc en particulier pour $x = x'$ on a pour tous les $j = 1, \ldots, n$:
- soit $c_j \cap \{x\} = \emptyset$ *i.e.* $x \notin c_j$ impossible pour x' et $j = i$ car $x' \in c_i \setminus v(c_i)$
- soit $x \in v(c_j)$ impossible pour x' et $j = i$ car $x' \in c_i \setminus v(c_i)$ (car par construction $c_i \setminus v(c_i)$ et $v(c_i)$ sont disjoints).

Donc S est parfaite.

($\Leftarrow$) Soit S une solution parfaite d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$, *i.e.* $\forall x \in S, \nexists i$ tel que $x \in c_i \setminus v(c_i)$ comme S est solution minimale *i.e.* $\forall x \in S, \exists i$ tel que $c_i \cap S = \{x\}$ on a donc que $\forall x \in S, \exists i$ tel que $v(c_i) \cap S = \{x\}$ puisque S est parfaite, donc S est préférée. Montrons maintenant qu'elle est préférée forte. S parfaite on a donc $\forall x \in S, \nexists i$ tel que $x \in c_i \setminus v(c_i)$ on obtient donc : $\forall x \in S, \forall i$
- soit $x \in c_i$ et alors $x \in v(c_i)$
- soit $x \notin c_i$

Donc S est préférée forte. □

2.3.4.2 Nouveaux critères "colorés"

Nous exploitons maintenant la coloration des strates, pour définir de nouveaux critères pour obtenir de "nouvelles" solutions.

Un bleu au moins Pour comparer deux solutions, nous utilisons le nombre d'éléments de la solution appartenant à B (formules au moins une fois bleues). La meilleure solution est celle qui touche le moins de formules au moins une fois bleue.

Définition 28 (Un bleu au moins) *Soit S_1, S_2 deux solutions d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$. $S_1 \leq_{B1f} S_2$ (on préfère S_2 à S_1) si et seulement si $|\{x \in S_1 / x \in B\}| \geq |\{x \in S_2 / x \in B\}|$* [5]

Exemple 16 $\mathcal{C} = \{\{a, b, \underline{d}\}, \{a, \underline{b}, e\}\}$ *Quelques solutions de faible cardinalité :*

$\underline{S_1} = \{a\}$
$\underline{S_2} = \{b\}$ ★
$S_3 = \{a, b\}$
$S_4 = \{a, d\}$ *optimale pour* $\leq_{B1f}$ *mais pas minimale pour l'inclusion*
$S_5 = \{a, e\}$
$S_6 = \{b, d\}$ *optimale pour* $\leq_{B1f}$ *mais pas minimale pour l'inclusion*
$S_7 = \{b, e\}$
$\underline{S_8} = \{d, e\}$

Remarque 21 *La solution optimale pour la relation $\leq_{B1f}$ n'est pas forcément minimale d'après l'exemple 16: S_6 est optimale pour la relation $\leq_{B1f}$ mais n'est pas minimale au sens de l'inclusion ensembliste.*

Théorème 5 *Soit S une solution minimale d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$, telle que $|\{x \in S / x \in B\}| = 0$. S est parfaite.*

Preuve : S'il existe une solution S d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$ telle que $|\{x \in S / x \in B\}| = 0$ *i.e.* $|\{x \in S / \exists i$ tel que $x \in c_i \setminus v(c_i)\}| = 0$ *i.e.* aucun élément de S n'appartient à $c_i \setminus v(c_i)$ d'un conflit donc forcément S est parfaite. □

[5] *i.e.* $|S_1 \cap B| \geq |S_2 \cap B|$

Un vert au moins Pour comparer deux solutions, nous utilisons le nombre d'éléments de la solution appartenant à V (formules au moins une fois vertes). La meilleure solution est celle qui touche le plus de formules au moins une fois verte.

Définition 29 (Un vert au moins) *Soit S_1, S_2 deux solutions d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$. $S_1 \leq_{V1f} S_2$ si et seulement si $|\{x \in S_1/x \in V\}| \leq |\{x \in S_2/x \in V\}|$*[6]

Exemple 17 $\mathcal{C} = \{\{\underline{a}, c\}, \{a, \underline{d}\}, \{b, \underline{d}\}, \{c, \underline{d}\}\}$ *Les solutions sont :*
$\underline{S_1} = \{a, b, c\}$ *optimale pour $\leq_{V1f}$ mais pas optimale pour $\leq_{B1f}$*
$\underline{S_2} = \{a, d\}$
$\underline{S_3} = \{c, d\}$ *optimale pour $\leq_{B1f}$ mais pas optimale pour $\leq_{V1f}$*

Exemple 17	S_1	S_2	S_3
B1f (def 28 page précédente)	3	1	1
V1f (def 29)	1	2	1

Remarque 22
- *Dans l'exemple 17 la solution S_3 optimale pour $\leq_{B1f}$ est différente de la solution S_1 optimale pour $\leq_{V1f}$.*
- *Dans l'exemple 16 page ci-contre, la solution S_4 est optimale pour la relation $\leq_{V1f}$ mais pas minimale pour l'inclusion.*
- *Pour pallier ce problème de minimalité nous pouvons considérer les mêmes critères $\leq_{B1f}$ et $\leq_{V1f}$ sur les solutions minimales uniquement.*

Présence Verte

Pour chaque formule d'une solution S nous totalisons sur tous les conflits le nombre de cette formule dans V.

Définition 30 (Pvert) *Soit $\mathcal{C} = \{c_i, i = 1..n\}$ un problème de conflits, nous définissons le nombre d'apparitions des éléments d'une solution S dans l'ensemble des modifiables (vert), de chaque conflit :*

$$Pvert(S) = \sum_i |\{x \in S/x \in v(c_i)\}|^7$$

Définition 31 (Présence verte) *Soit S_1, S_2 deux solutions d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$. $S_1 \leq_{Pvert} S_2$ si et seulement si $Pvert(S_1) \leq Pvert(S_2)$.*

Remarque 23 *Dans l'exemple 16 page précédente, la solution S_6 est optimale pour la relation $\leq_{Pvert}$ mais pas minimale pour l'inclusion. Pour pallier ce problème de minimalité nous pouvons appliquer le même critère sur les solutions minimales uniquement.*

Présence Bleue

De la même manière que nous avons introduit Pmax et Pvert, nous définissons Pbleu.

Définition 32 (Pbleu) *Soit $\mathcal{C} = \{c_i, i = 1..n\}$ un problème de conflits, nous définissons le nombre d'apparitions des éléments d'une solution S dans l'ensemble des éventuellement modifiables (bleus) de chaque conflit:*

$$Pbleu(S) = \sum_i |\{x \in S/x \in (c_i \setminus v(c_i))\}|^8$$

Corollaire 2 *Si S une solution minimale d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$ telle que : $Pbleu(S) = 0$ alors S est parfaite.*

Preuve : Soit S une solution minimale d'un problème de conflits 2-stratifié $\mathcal{C} = \{c_i, i = 1..n\}$ telle que :

$$Pbleu(S) = 0 \Leftrightarrow \sum_i |\{x \in S/x \in (c_i \setminus v(c_i))\}| = 0$$

Ce qui signifie qu'il n'y a jamais d'élément de la solution coloré en bleu

[6] *i.e.* $|S_1 \cap V| \geq |S_2 \cap V|$
[7] *i.e.* $|S \cap v(c_i)|$
[8] *i.e.* $|S \cap b(c_i)|$

$$\Rightarrow |\{x \in S/x \in B\}| = 0$$

D'après le théorème 5 page 24, S est parfaite. □

Théorème 6 *Pour chaque solution S d'un problème de conflits 2-stratifié $\mathcal{C} = \{c_i, i = 1..n\}$*

$$Pmax(S) = Pbleu(S) + Pvert(S)$$

i.e.

$$\sum_i |\{x \in S/x \in (c_i \setminus v(c_i))\}| + \sum_i |\{x \in S/x \in v(c_i)\}| = \sum_i |\{S \cap c_i\}|$$

Preuve : D'après les définitions de Présence Bleu et de Présence Vert

$$Pvert(S) = \sum_i |\{x \in S/x \in v(c_i)\}|$$

$$Pbleu(S) = \sum_i |\{x \in S/x \in (c_i \setminus v(c_i))\}|$$

$$Pbleu(S) + Pvert(S) = \sum_i |\{x \in S/x \in (c_i \setminus v(c_i))\}| + \sum_i |\{x \in S/x \in v(c_i)\}|$$

Les deux ensembles sur lesquels la somme est effectuée sont disjoints par construction, donc on regroupe les termes pour obtenir

$$Pbleu(S) + Pvert(S) = \sum_i |\{x \in S/x \in (c_i \setminus v(c_i))\}| + |\{x \in S/x \in v(c_i)\}|$$

Comme la coloration forme une partition pour chaque c_i, on obtient :

$$Pbleu(S) + Pvert(S) = \sum_i |\{x \in S \cap c_i\}|$$

Donc $Pbleu(S) + Pvert(S) = Pmax(S)$ □

Définition 33 (Présence bleue) *Soit S_1, S_2 deux solutions d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$. $S_2 \leq_{Pbleu} S_1$ si et seulement si $Pbleu(S_1) \leq Pbleu(S_2)$.*

Exemple 18 $\mathcal{C} = \{\{b, \underline{a}\}, \{a, \underline{d}\}, \{b, \underline{c}\}, \{c, \underline{d}\}, \{a, c, \underline{f}\}, \{g, \underline{a, c}\}, \{c, \underline{a, e}\}\}$

$\underline{S_1} = \{a, c\}$★
$\underline{S_2} = \{b, d, a\}$
$\underline{S_3} = \{b, d, c\}$
$\underline{S_4} = \{b, d, e, f, g\}$

Critère	S_1	S_2	S_3	S_4
V1f (max) (def 29 page précédente)	*2*	*2*	*2*	*3*
B1f (min) (def 28 page 24)	*2*	*2*	*2*	*2*
Pmax (def 19 page 19)	*10*	*9*	*9*	*7*
Pvert(max)(def 31 page précédente)	*5*	*5*	*4*	*4*
Pbleu(min)(def 33)	*5*	*4*	*5*	*3*

Calcul des différents critères

Nous remarquons que même sur deux solutions minimales S_4 et S_1 on a $S_4 \leq_{Pvert} S_1$ mais $S_4 \not\leq_{Pbleu} S_1$

Corollaire 3 *La présence verte et la présence bleue sont deux relations équivalentes, pour des solutions S_1 et S_2 telles $Pmax(S_1) = Pmax(S_2)$ i.e. Si $Pmax(S_1) = Pmax(S_2)$ alors $S_1 \leq_{Pbleu} S_2 \Leftrightarrow S_1 \leq_{Pvert} S_2$.*

Preuve : D'après le théorème 6 on a :

– $Pmax(S_1) = Pbleu(S_1) + Pvert(S_1)$
– $Pmax(S_2) = Pbleu(S_2) + Pvert(S_2)$
Or $Pmax(S_1) = Pmax(S_2)$
Donc $Pbleu(S_1) + Pvert(S_1) = Pbleu(S_2) + Pvert(S_2)$
Donc $Pvert(S_1) = Pbleu(S_2) + Pvert(S_2) - Pbleu(S_1)$
$S_1 \leq_{Pvert} S_2 \Leftrightarrow Pvert(S_1) \leq Pvert(S_2)$
$S_1 \leq_{Pvert} S_2 \Leftrightarrow Pbleu(S_2) + Pvert(S_2) - Pbleu(S_1) \leq Pvert(S_2)$
$S_1 \leq_{Pvert} S_2 \Leftrightarrow Pbleu(S_2) - Pbleu(S_1) \leq 0$
$S_1 \leq_{Pvert} S_2 \Leftrightarrow Pbleu(S_2) \leq Pbleu(S_1)$
$S_1 \leq_{Pvert} S_2 \Leftrightarrow S_1 \leq_{Pbleu} S_2$
□

Bilan
Dans l'exemple 16 page 24, la solution S_6 est optimale pour la relation $\leq_{Pbleu}$ mais pas minimale pour l'inclusion. Donc $\leq_{Pbleu}$ et $\leq_{Pvert}$ ne respectent pas le principe de minimalité par rapport à l'inclusion. On peut pour pallier cette lacune, poser $S_1 \leq_{inclPbleu} S_2$ si et seulement si
– S_1 et S_2 minimales.
– $S_1 \leq_{Pbleu} S_2$
Nous avons d'autres possibilités pour associer $\leq_{Pbleu}$ à d'autres critères non colorés :
– Pbleu puis la parcimonie.
– La parcimonie puis Pbleu pour départager les solutions *ex-aequo.*
– *Pbleu puis la minimalité.*
– *La minimalité puis Pbleu pour départager les solutions ex-aequo.*

2.3.5 Propriétés des différents critères

En annexe B page 51 pour chaque exemple du chapitre nous donnons la solution optimale suivant chaque critère.

Définition 34 (Score) *Soit S l'ensemble des solutions d'un problème P d'ordonnancement. On appelle Score une application qui, à chaque solution S_i de S, associe un entier.*

$$Score : S_i \rightarrow \mathbb{N}$$

Définition 35 ($\leq_{score}$) *Une relation de préférence $\leq_{score}$ est définie à partir d'un score :*

$$S_1 \leq_{score} S_2 \Leftrightarrow Score(S_1) \geq Score(S_2)$$

Propriété 10 *Toute relation de préférence définie à partir d'un score est transitive.*

Preuve : Score est une application à valeurs dans $\mathbb{N}$, $\leq_{score}$ correspond à la relation $\leq$ habituelle de $\mathbb{N}$, qui est évidemment transitive donc $\leq_{score}$ l'est. □

Corollaire 4 *Les relations de parcimonie, d'un vert au moins, d'un bleu au moins, de présence Max, de présence Vert et de présence Bleu sont transitives.*

Preuve : Les relations de parcimonie, d'un vert au moins, d'un bleu au moins, de présence Max, de présence Vert, de présence Bleu, sont définies par des Scores.

Critère	Score associé
Parcimonie (def 17 page 18)	$\|S\|$
V1f (max) (def 29 page 25)	$\|\{x \in S/x \in V\}\|$
B1f (min) (def 28 page 24)	$\|\{x \in S/x \in B\}\|$
Pmaxmin(def 20 page 20)	$\sum_i \|\{x \in S \cap c_i\}\|$
Pvert(max)(def 31 page 25)	$\sum_i \|\{x \in S/x \in v(c_i)\}\|$
Pbleu(min)(def 33 page ci-contre)	$\sum_i \|\{x \in S/x \in (c_i \setminus v(c_i))\}\|$

D'après la propriété 10 ces relations sont transitives.
□

Critère	minimalité	parcimonie	transitif	quasi-transitivité
Minimal (def 16 page 17)	oui	non	oui	oui
Min Forte (def 18 page 18)	oui	non	oui	oui
Card (min)(def 17 page 18)	oui	oui	oui	oui
V1f (max) (def 29 page 25)	non	non	oui	oui
B1f (min) (def 28 page 24)	non	non	oui	oui
Pref (def 22 page 21)	oui	non	oui	oui
Pref-F (def 27 page 23)	oui	non	oui	oui
Pmaxmin(def 20 page 20)	oui	non	oui	oui
Pvert(max)(def 31 page 25)	non	non	oui	oui
Pbleu(min)(def 33 page 26)	non	non	oui	oui
eli (def 23 page 21)	oui	non	non	non
eli-F (def 24 page 22)	oui	non	non	non

TAB. 2.1: Récapitulatif des propriétés des différents critères.

Remarque 24 *La relation $\geq_{incl}$ selon la définition 16 page 17 est naturellement transitive mais pas complète.*

Nous récapitulons dans le tableau 2.1 les différentes propriétés de nos critères. Un critère vérifie la minimalité si toutes les optimales solutions pour ce critère sont minimales (au sens de l'inclusion), de même pour la parcimonie.

2.4 Conclusion

Nous avons dégagé des critères en adaptant ceux existant. Nous avons également mis en évidence de nouveaux critères basés sur le principe de coloration.

Nous ne considérerons que les critères transitifs, car un critère, qui n'est pas transitif, "trouble" la notion d'ordre qui est généralement associe à un critère. Pour les critères sans préférence, essayer de résoudre un conflit en enlevant le moins d'éléments possibles (solution parcimonieuse) semble être "naturel". Car trouver une solution parcimonieuse correspond à un critère transitif qui produit une solution minimale au sens de l'inclusion. Pour les critères avec préférence, nous sélectionnerons un critère qui donne une solution la plus "proche" de la solution parfaite (cf définition 25 page 23). D'après les différentes propriétés étudiées, les critères qui minimisent le nombre de "Bleus" sont ceux qui sont capables de fournir une solution parfaite si elle existe. Ces critères sont transitifs, mais pas minimaux au sens de l'inclusion d'ensembles. Comme nous l'avons souligné il faudra introduire, dans ces critères colorés, la minimalité par rapport à l'inclusion, pour obtenir des solutions minimales, ce qui semble être implicitement souhaité par l'utilisateur.

Nous allons maintenant voir comment trouver une solution dans un problème d'ordonnancement selon les différents critères transitifs. Une fois ces algorithmes de recherche de solutions trouvés, nous regarderons comment relâcher une contrainte présente dans une solution. Enfin nous implémenterons la recherche de solutions parcimonieuses dans un problème de conflit.

Chapitre 3

Méthodes classiques issues de la "logique" pour résoudre les conflits

Dans ce chapitre, nous cherchons une ou des solutions pour un problème d'ordonnancement inconsistant sachant que l'utilisateur a exprimé des préférences locales sur l'ensemble des conflits. Les solutions recherchées devront satisfaire au mieux les préférences de l'utilisateur au sens des critères évoqués au chapitre 2. Nous présentons d'abord une technique pour résoudre les conflits basée sur l'énumération des éléments des conflits, inspirée par l'algorithme de Davis et Putnam [DP60] développé pour prouver la satisfiabilité d'une base de clauses. Cette méthode est issue de travaux portant sur le problème dit de "hitting set" . Nous allons reprendre les algorithmes de type "Davis et Putnam" proposés par Castell et Leberre dans [Cas96, Leb00] en les adaptant à la recherche de solution pour les conflits d'un problème d'ordonnancement. Nous apportons une amélioration en coupant certaines branches de l'arbre de recherche ceci, grâce à la technique dite du "Branch-and-Bound" présentée dans [LW66], utilisée dans [BKSO96]. Enfin nous établissons pour chacun de nos critères les algorithmes de "Branch-and-Bound" associés.

3.1 Résolution par énumération de toutes les solutions

Soit l'ensemble des conflits minimaux $\mathcal{C} = \{c_i, i = 1, \dots, p\}$, pour trouver une solution optimale au sens d'un critère choisi, deux approches différentes se dessinent :
- prendre dans chaque conflit c_i un élément pour construire toutes les solutions, méthode proposée dans [Rei87, GSW89, Lev92].
- énumérer tous les sous-ensembles de l'ensemble $N = \cup_i c_i$ des éléments. Davis et Putnam ont développé cette idée dans [DP60]. La technique utilisée consiste à prendre ou à ne pas prendre un élément de N dans la solution. Cette méthode donne tous les sous-ensembles existants de l'ensemble des conflits. Cette technique fournit des ensembles d'éléments qui ne seront pas forcément des solutions (car ils ne rencontrent pas toujours tous les conflits).

Nous choisissons de construire les solutions en adaptant les algorithmes de type "Davis et Putnam" développés par Castell pour le calcul d'un modèle dans une base de clauses et par Leberre pour calculer d'impliquants P-restreints. Dans leurs approches respectives, ils utilisent des tests d'arrêt de recherche lorsque le résultat est trouvé ou bien lorsqu'il est certain que l'on ne trouvera pas de solution. Ensuite ils effectuent un branchement, "spliting rule", pour considérer les deux cas de figure qui se présentent : soit on choisit la clause courante soit on la rejette. On dégage grâce ces travaux des "règles pratiques" pour la construction des solutions pour nos conflits:
- **Tests d'arrêt**
  - Si l'ensemble des conflits est vide alors on a construit une solution.
  - Si un conflit est vide *i.e.* il n'y a plus aucun élément qui puisse le résoudre, on arrête l'exploration dans cette branche qui ne fournira pas de solution.
- **Règle de branchement**
  - Si x est sélectionné pour la construction de S, on élimine alors tous les conflits contenant x.

- Si x est exclu de la construction, on supprime toutes les occurrences de x dans tous les conflits restant à traiter.

L'algorithme 6 basé sur les travaux de Castell et Leberre respecte les principes énoncés et calcule toutes les solutions . On lance Trouver-DP($\mathcal{C}$,$\mathcal{S}$ avec tous les conflits $\mathcal{C} = \{c_i, i = 1..n\}$ et la solution courante $Sol = \{\}$. Il faut également initialiser la variable globale $\mathcal{S} = \{\}$.

Algorithme 6: Trouver-DP($\mathcal{C}$: conflits,Sol : Solution courante)

début
- **si** $|\mathcal{C}| = 0$ **alors**
  - Ajouter(Sol,$\mathcal{S}$);
- **sinon**
  - **si** *il n'existe pas de conflit vide dans* $\mathcal{C}$ **alors**
    - $x = PrendreElement(\mathcal{C})$;
    - % IN : on met x dans la solution et on propage après nettoyage des conflits%
    - Trouver-DP(Nettoyage($\mathcal{C}$,x),$Sol \cup \{x\}$);
    - % OUT : on ne met pas x dans la solution et on propage après suppression de x des conflits restant%
    - Trouver-DP(Supprimer($\mathcal{C}$,x),Sol);
  - % sinon coupure de la branche %

fin

Procedure `Supprimer(`$\mathcal{C}$ *: conflits,* x *: élément*`)`

début
- **pour** *Tous les conflits* c_i *de* $\mathcal{C}$ **faire** $c_i = c_i \setminus x$;
- retourner($\mathcal{C}$);

fin

Procedure `Nettoyage(`$\mathcal{C}$ *: ensemble de conflits ; x: élément*`)`

début
- **pour** *Tous les conflits* c_i *de* $\mathcal{C}$ **faire**
  - **si** $x \in c_i$ **alors** $\mathcal{C} = \mathcal{C} \setminus c_i$;
- % c_i est résolu%
- retourner ($\mathcal{C}$);

fin

La procédure $PrendreElement$ détermine l'heuristique de recherche des solutions.

Remarque 25
- *Comme l'a souligné [Cas97] un conflit qui ne contient qu'un seul élément doit faire partie de la solution donc autant l'introduire dès que possible pour éviter d'avoir à refaire ce choix dans de nombreuses branches de l'arbre de recherche.*
- *Choisir l'élément le plus présent dans les conflits, permet de résoudre de nombreux conflits d'un seul coup. Cette heuristique peut accélérer la recherche de la première solution mais risque de construire un arbre de recherche "complexe".*

3.2 Algorithme classique de "Branch-and-Bound"

L'algorithme 7 page ci-contre est une version adaptée à l'optimisation de l'algorithme 6, on met ou on ne met pas un élément d'un conflit dans la solution. De plus, on coupe les branches dont la solution courante dépasse une certaine borne, au fur et à mesure de l'élaboration de la solution. C'est un algorithme de type "Branch-and-Bound" qui calcule les solutions optimales suivant le critère choisi, sous réserve qu'il soit transitif. A chaque critère correspond un test de coupure et des mises à jour des bornes.

Pour un problème de conflits $\mathcal{C} = \{c_i, i = 1, \ldots, p\}$ on lance Initialisation-Borne(*Borne*) puis Trouver-solution($\mathcal{C}$,{},0) avec l'ensemble des solutions suivant $\mathcal{S} = \{\}$ en variable globale.

Algorithme 7: Trouver-solution($\mathcal{C}$: ensemble de conflits ; *Sol* : solution; *Bornecourante* : entier)

```
début
    si |C| = 0 alors
        % S solution de C %
        Mise-a-jour-B(Borne,Bornecourante);
        Ajouter(Sol,S);
    sinon
        si non(Test-coupure(Sol, Borne, Bornecourante)) et (il n'existe pas de conflit vide dans C)
        alors
            x = PrendreElement(C);
            % IN : on met x dans la solution et on propage après nettoyage des conflits%
            Trouver-solution(Nettoyage(C,x),Sol ∪ {x},Mise-a-jour-BC(Bornecourante,x,C, Sol));
            % OUT : on ne met pas x dans la solution et on propage après suppression de x des conflits
            restant%
            Trouver-solution(Supprimer(C,x),Sol,Bornecourante);
    % sinon coupure de la branche %
fin
```

Pour chaque un critère (parcimonie, ...) il faut définir explicitement les fonctions du "Branch-and-Bound" de l'algorithme 7 :

- Initialisation-Borne(*Borne*);
  Initialise la variable globale *Borne*.
- Mise-a-jour-B(*Borne*,*Bornecourante*);
  Lorsqu'une nouvelle solution est trouvée on met à jour la *Borne*.
- Test-coupure(S,*Borne*,*Bornecourante*);
  En fonction des solutions cherchées, de la solution courante et des valeurs des bornes on décide de couper la branche ou de continuer l'exploration.
- Mise-a-jour-BC (*Bornecourante*,x,$\mathcal{C}$, S);
  On actualise la *Bornecourante* au fur et à mesure de la construction de la solution, pour ne pas perdre d'information.

Nous donnerons explicitement dans la section 3.3 toutes ces fonctions pour des algorithmes correspondants à la plupart des critères.

Remarque 26

- *Seule "la" dernière solution fournie par un tel principe, sera optimale selon le critère choisi.*
- *Bien initialiser la Borne permet une exploration moins profonde pour trouver la première solution.*
- *Un algorithme de type Branch-and-Bound parcourant l'arbre de recherche en profondeur d'abord qui atteint en premier une solution proche de la solution cherchée, permet d'obtenir une "bonne" première valeur de la Borne, ce qui réduit considérablement l'arbre de recherche.*

3.3 Spécification d'algorithmes pour trouver les solutions optimales selon les différents critères du chapitre 2

3.3.1 Solution minimale

Définition 36 (Ensemble subsumé) *Un ensemble S est subsumé par un ensemble S' si et seulement si $S' \subseteq S$.*

Pour obtenir les solutions minimales, nous utilisons l'algorithme 6 page 30 auquel nous apportons quelques modifications.

- Avant de $PrendreElement$ regarder si la solution courante Sol ne contient pas une solution existante *i.e.* une solution $s \in S$ telle que $s \subseteq Sol$ et $s \neq \{\}$. Si c'est le cas, il faut couper la branche.
- De plus, lors de l'ajout de solution il faut ôter toutes les solutions déjà présentes qui sont subsumées par la nouvelle solution.

Remarque 27 *Nous pouvons aussi adapter un algorithme de type "Branch-and-Bound" pour qu'il trouve les solutions minimales ceci sans utiliser de "Borne". Il suffit de prendre en compte les points suivants :*

- *Lorsque l'on trouve une nouvelle solution : toutes les solutions qui peuvent être subsumées par la nouvelle solution sont enlevées de l'ensemble des solutions.*
- *Test-coupure : s'il existe une solution de l'ensemble des solutions qui subsume la solution courante, on coupe alors la branche.*

Nous utiliserons cette remarque pour que les critères colorés nous fournissent des solutions minimales au sens de l'inclusion ensembliste et colorées. Comme le fait remarquer Castell, supprimer les solutions subsumées est une opération coûteuse et elle risque d'être plus coûteuse que l'amélioration qu'elle apporte.

3.3.2 Solution minimale forte

Rappelons que la solution minimale forte est la solution qui ne possède qu'un élément dans chaque conflit (définition 16 page 17). Dans l'algorithme 6 page 30, qui calcule les solutions minimales, on remplace la fonction nettoyage par la fonction nettoyage-Fort. Cette fonction supprime tous les conflits où apparaît le x choisi et toutes les occurrences des autres éléments du conflit où x est présent. Ainsi chaque élément de chaque solution générée sera présent une seule fois dans chaque conflit. S'il n'y a pas de solution minimale forte, l'algorithme ne trouvera pas de solution.

Procedure `nettoyage-Fort`($\mathcal{C}$ *: conflits,* $conf$ *: conflit,* x *: élément*)

---

début
  % On supprime les conflits où x apparaît et on mémorise dans M toutes les occurrences des autres éléments des conflits où x est présent %
  pour *Tous les conflits* c_i *de* $\mathcal{C}$ **faire**
    si $x \in c_i$ **alors**
      $\mathcal{C} = \mathcal{C} \setminus c_i$
      $M = M \cup (c_i \setminus \{x\})$
  % On enlève de tous les conflits restant tous les éléments de M %
  tant que $M \neq \emptyset$ **faire**
    $y = first(M)$;
    $\mathcal{C} =$ Nettoyage($\mathcal{C} \setminus conf, y$);
    $M = M \setminus \{y\}$;
  retourner($\mathcal{C} \setminus conf$);
fin

3.3.3 Solution parcimonieuse

Rappelons que la solution parcimonieuse est la solution qui a le plus petit nombre d'éléments (définition 17 page 18).

Nous allons utiliser un algorithme de type "Branch-and-Bound" dans lequel on explicite les fonctions :

- Initialisation-Borne($Borne$);
  $Borne = |\mathcal{C}|$; % Au maximum il y a un élément de chaque conflit dans une solution.%
- Mise-a-jour-B($Borne$,$Bornecourante$);
  Si $|S| \leq Borne$ alors $Borne = |S|$; % Le cardinal de la solution trouvée devient la nouvelle borne%

- Test-coupure(S, $Borne$, $Bornecourante$);
  $Bornecourante > Borne$; % Si la $Bornecourante$ dépasse la $Borne$ il faut s'arrêter%
- Mise-a-jour-BC ($Bornecourante$,x,$\mathcal{C}$, S);
  Si $x \notin S$ $Bornecourante = Bornecourante + 1$;

L'algorithme "Branch-and-Bound" utilisant les fonctions ci-dessus calcule toutes les solutions parcimonieuses par coupure des branches qui dépassent le nombre d'éléments des solutions déjà trouvées.

Remarque 28 *Étant donné que nous cherchons la solution parcimonieuse i.e. solution de plus petit cardinal, le test de coupure devra porter sur le cardinal de la solution courante.*

Remarque 29 *Classer les conflits par ordre croissant suivant le nombre d'éléments les constituant accélère la recherche.*

3.3.4 Solution optimale pour $\leq_{V1f}$

Rappelons que la solution "verte au moins une fois" est la solution qui possède le maximum d'éléments "au moins une fois" parmi les "Verts" (on maximise le score : $|\{x \in S/x \in V\}|$ où $V = \cup_i v(c_i)$ cf définition 29 page 25).

Utilisons un algorithme de type "Branch-and-Bound" , en explicitant les fonctions suivantes :

- Initialisation-Borne($Borne$);
  $Borne = 0$;
- Mise-a-jour-B($Borne$,$Bornecourante$);
  Si $|S \cap V| \geq Borne$ alors $Borne = |S \cap V|$; % Il faut maximiser le nombre d'éléments au moins une fois Vert%
- Test-coupure(S, $Borne$, $Bornecourante$);
  $Bornecourante + |\mathcal{C}| < Borne$; % Au mieux on relâchera au maximum $|\mathcal{C}|$ verts différents%
- Mise-a-jour-BC ($Bornecourante$,x,$\mathcal{C}$, S);
  Si $x \in V$ et $x \notin S$ $Bornecourante = Bornecourante + 1$;

Remarque 30 *L'ensemble V est une solution donnée par cet algorithme qui n'est pas forcément minimale. Pour obtenir des solutions minimales optimales pour $\leq_{V1f}$, il suffit de modifier légèrement l'algorithme suivant la remarque 27 page ci-contre.*

3.3.5 Solution optimale pour $\leq_{B1f}$

Rappelons que la solution bleue "au moins une fois" est la solution qui possède le minimum d'éléments "au moins une fois" parmi les "Bleus" (on minimise le score : $|\{x \in S/x \in B\}|$ où $B = \cup_i b(c_i)$ cf définition page 24). Nous allons utiliser un algorithme de type "Branch-and-Bound":

- Initialisation-Borne($Borne$);
  $Borne = |\mathcal{C}|$; % Au pire il y a un élément de chaque conflit dans une solution et ils sont tous bleus.%
- Mise-a-jour-B($Borne$,$Bornecourante$);
  Si $Bornecourante \leq Borne$ alors $Borne = Bornecourante$; % Il faut minimiser le nombre d'éléments au moins une fois Bleu%
- Test-coupure(S, $Borne$, $Bornecourante$);
  $Bornecourante > Borne$; % Si l'état de la solution courante est plus grand que la borne autant s'arrêter%
- Mise-a-jour-BC ($Bornecourante$,x,$\mathcal{C}$, S);
  Si $x \in B$ et $x \notin S$ $Bornecourante = Bornecourante + 1$;

Remarque 31 *Pour obtenir des solutions minimales optimales pour $\leq_{B1f}$ il suffit de modifier légèrement l'algorithme suivant la remarque 27 page ci-contre.*

3.3.6 Solution minimale préférée

On cherche à trouver les solutions minimales préférées (définition 22 page 21). On améliore l'algorithme 6 page 30 qui calcule les solutions minimales pour qu'il trouve les solutions préférées. Dans l'algorithme, au lieu de prendre le premier élément de N on va choisir un élément de V (V = ensemble des éléments verts dans les conflits restant) *i.e.* on ne relâche que des verts "au moins une fois". Ainsi les solutions construites ne contiennent forcément que des "verts".

On remplace $x = first(N)$; par $x = first(V)$;
Et c'est V qui évolue et plus N. D'après la propriété 7 page 21 si les conflits sont tous 2-stratifiés avec 2 strates non vides par conflit, alors il existe une solution minimale préférée. On est sûr de trouver une solution minimale préférée car dès que $V = \emptyset$ tous les conflits sont résolus (même s'ils peuvent l'être avant). L'algorithme fournit bien les solutions minimales préférées.

3.3.7 Solution minimale fortement préférée

Il faut maintenant que les éléments de la solution n'appartiennent qu'à l'ensemble des verts (définition 27 page 23). On remplace $x = first(N)$; par $x = first(V \setminus B)$; . D'après la remarque 11 page 19 il peut ne pas exister de solution minimale préférée forte. On utilise le même principe que dans l'algorithme nettoyage-Fort page 32 pour la minimalité forte : tant que $V \setminus B \neq \emptyset$ *i.e.* on ne relâche que des verts qui ne sont jamais bleus. S'il reste des conflits alors il faut stopper et jeter la solution trouvée (qui n'en est pas une) sinon la solution trouvée est minimale préférée forte (donc parfaite).

3.3.8 Solution optimale pour $\leq_{Pmaxmin}$

Rappelons que la solution optimale selon $\leq_{Pmaxmin}$ (définition 20 page 20) est la solution dont les éléments apparaissent le plus petit nombre de fois dans les conflits (*i.e.* minimisation du score $Pmax(S) = \sum_i |S \cap c_i|$). Nous allons utiliser un algorithme de type "Branch-and-Bound" :

- Initialisation-Borne($Borne$);
  $Borne = |\mathcal{C}|$; % Au pire il y a un élément de chaque conflit dans une solution.%
- Mise-a-jour-B($Borne$,$Bornecourante$);
  Si $Bornecourante \leq Borne$ alors $Borne = Bornecourante$; % Il faut compter le nombre d'apparitions des éléments de la solution ceci grâce à la $Bornecourante$%
- Test-coupure(S, $Borne$, $Bornecourante$);
  $Bornecourante > Borne$; % Si l'état de la $Bornecourante$ dépasse la $Borne$ alors la solution construite aura plus d'apparitions que la solution déjà trouvée, autant stopper la recherche %
- Mise-a-jour-BC ($Bornecourante$,x, $\mathcal{C}$, S);
  $Bornecourante = Bornecourante +$ Nombre d'apparitions de x dans les conflits de $\mathcal{C}$;

Il faut faire l'appel de la procédure avec $Bornecourante = 0$;

3.3.9 Solution optimale pour $\leq_{Pvert}$

Rappelons que la solution optimale selon $\leq_{Pvert}$ est la solution dont les éléments apparaissent le plus grand nombre de fois "en vert" dans les conflits (on minimise le score de $Pvert(S) = \sum_i |\{x \in S\}/x \in v(c_i)\}|$ cf définition 31 page 25).

Nous allons utiliser un algorithme de type "Branch-and-Bound" :

- Initialisation-Borne($Borne$);
  $Borne = 0$; %la $Borne$ va croître %
- Mise-a-jour-B($Borne$,$Bornecourante$);
  Si $Borne \leq Bornecourante$ alors $Borne = Bornecourante$; % Il faut compter le nombre d'apparitions en vert des éléments de la solution ceci grâce à la $Bornecourante$%
- Test-coupure(S, $Borne$, $Bornecourante$);
  $Bornecourante + |\mathcal{C}| < Borne$; % Si l'état de la $Bornecourante$ plus le maximum de conflits restant en supposant qu'ils soient tous résolus par des verts est plus petit que la $Borne$ alors la solution construite aura moins d'éléments apparus en vert que la solution déjà trouvée%
- Mise-a-jour-BC ($Bornecourante$,x, $\mathcal{C}$, S);
  $Bornecourante = Bornecourante +$ Nombre d'apparitions dans $\mathcal{C}$ de x en tant que vert *i.e.* on ajoute 1 pour tous les conflits c_i tels que $x \in V \cap c_i$;

Il faut faire l'appel de la procédure avec $Bornecourante = 0$;

Remarque 32 *Pour obtenir des solutions minimales optimales pour $\leq_{Pvert}$ il suffit de modifier légèrement l'algorithme suivant la remarque 27 page 32*

3.3.10 Solution optimale pour $\leq_{PBleu}$

Rappelons que la solution optimale selon $\leq_{PBleu}$ est la solution dont les éléments apparaissent le plus petit nombre de fois "en bleu" dans les conflits (on minimise le score $Pbleu(S) = \sum_i |\{x \in S/x \in (c_i \setminus v(c_i))\}|$ cf définition 33 page 26). Nous allons utiliser un algorithme de type "Branch-and-Bound" :

- Initialisation-Borne($Borne$);
  $Borne = |\mathcal{C}|$; % Au maximum il y a un élément de chaque conflit dans une solution.%
- Mise-a-jour-B($Borne$,$Bornecourante$);
  Si $Bornecourante \leq Borne$ alors $Borne = Bornecourante$; % Il faut compter le nombre d'apparitions en bleu des éléments de la solution ceci grâce à la $Bornecourante$%
- Test-coupure(S, $Borne$, $Bornecourante$);
  $Bornecourante > Borne$; % Si l'état de la $Bornecourante$ dépasse la $Borne$ alors la solution construite aura plus d'éléments apparus en bleu que la solution déjà trouvée%
- Mise-a-jour-BC ($Bornecourante$,x,$\mathcal{C}$, S);
  $Bornecourante = Bornecourante +$ Nombre d'apparitions dans $\mathcal{C}$ de x en tant que bleu *i.e.* on ajoute 1 pour tous les conflits c_I tels que c_i $x \in B \cap c_i$;

Il faut faire l'appel de la procédure avec $Bornecourante = 0$;

Remarque 33 *Pour obtenir des solutions minimales optimales pour $\leq_{Pbleu}$ il suffit de modifier légèrement l'algorithme suivant la remarque 27 page 32*

3.4 Conclusion

Les algorithmes de Castell et Leberre, basés sur l'idée de Davis et Putnam, adaptés et améliorés par la coupure de branches permettent de trouver pour chacun de nos critères transitifs un algorithme de recherche de solutions. Il est également possible d'envisager pour chaque critère une heuristique permettant un choix plus approprié de l'élément à tester. Par contre, les critères non-transitifs semblent plus difficiles à mettre en place, nous ne les avons pas étudiés.

Dans cette approche "logique" de résolution des conflits, on calcule les conflits et ensuite on cherche une solution. En partant des conflits, on oublie la nature de notre problème d'ordonnancement (la "topologie" du graphe associé). Nous cherchons dans le chapitre 4 à appliquer ces méthodes de résolution de conflits tout en prenant en compte les spécificités des problèmes d'ordonnancement de tâches.

Chapitre 4

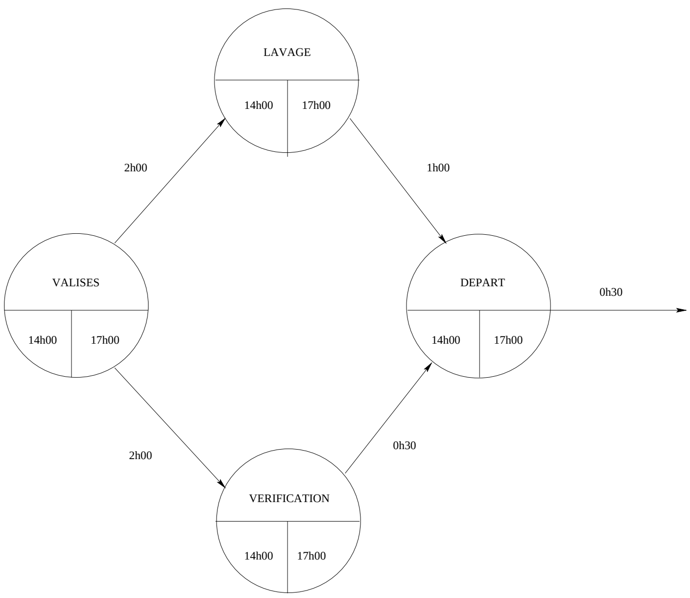
Résolution des conflits par modification des contraintes dans un PERT

Après avoir présenté les algorithmes de résolution de conflits issus des bibliothèques d'algorithmes de logique propositionnelle, nous allons chercher à adapter ces méthodes au PERT. Remarquons que jusqu'à présent les formules sélectionnées pour résoudre les conflits disparaissaient, elles étaient supprimées.
Dans un problème d'ordonnancement, les "formules" sont des contraintes. Nous avons plus de possibilités qu'en logique, nous pouvons :

- soit supprimer la contrainte *i.e.* enlever la contrainte du système d'inéquations.
- soit relaxer une constante associée à la contrainte *i.e.* modifier la valeur de la constante et conserver la contrainte, c'est-à-dire si la contrainte est :
  - une contrainte de livraison : la date de livraison associée passe à une valeur "infinie". On élimine donc tous les conflits contenant cette date de livraison.
  - une contrainte de disponibilité : la date de disponibilité associée prend une grande "valeur négative". On élimine ainsi tous les conflits créés par cette contrainte de disponibilité.
  - une contrainte de précédence : on met la durée à 0.

  Si on veut supprimer une tâche : on supprime ou modifie toutes les contraintes associées. Observons sur l'exemple 19 les différences entre ces alternatives.

Exemple 19 *M. et Mme Holiday partent en vacances pour cela ils doivent faire les valises ensemble, l'un lavera la voiture pendant que l'autre vérifiera la maison et enfin ils partiront ensemble avec leurs bagages en vacances.*



Il y a un chemin inconsistant :
– $r_{VALISES} = 14h00$
– $t_{VALISES} \leq r_{VALISES}$
– $d_{VALISES} = 2h00$
– $t_{LAVER} \geq t_{VALISES} + d_{VALISES}$
– $d_{LAVER} = 1h00$
– $t_{PARTIR} \geq t_{LAVER} + d_{LAVER}$
– $d_{PARTIR} = 0h30$
– $l_{PARTIR} = 17h00$
– $l_{PARTIR} \leq t_{PARTIR} + d_{PARTIR}$

– Modifier une disponibilité r_i affecte tous les chemins commençant par r_i. Par exemple, passer $r_{VALISES}$ à $13h30$ rétablit la consistance et modifie l'autre chemin.
– Modifier une livraison touche tous les chemins finissant en l_i. Dans cet exemple, passer l_{PARTIR} à $17h30$ résout le conflit et modifie également l'autre chemin critique qui devient alors consistant.
– Changer une précédence :
  – Modifier une durée d_i touche tous les chemins contenant d_i. Si on passe $d_{VALISES}$ à $1h30$ ou d_{PARTIR} à 0 cela résout le conflit et modifie également l'autre chemin critique qui devient alors consistant.
  – Ôter une précédence touche tous les chemins la contenant. Par exemple si on ôte $t_{LAVER} \geq t_{VALISES} + d_{VALISES}$ alors la tâche $LAVER$ n'a plus de précédence et le conflit est résolu. Si on ôte $t_{PARTIR} \geq t_{LAVER} + d_{LAVER}$ le conflit est résolu et la tâche $LAVER$ n'a plus de successeur.
– Ôter une tâche c'est enlever toutes les équations associées à cette tâche.

Observons que si les relations de précédence ne sont pas explicitement données par l'utilisateur la suppression d'une précédence peut poser quelques problèmes. Imaginons que la tâche de $VERIFICATION$ n'existe plus. Si l'on décide de supprimer la précédence $t_{LAVER} \geq t_{VALISES} + d_{VALISES}$ pour résoudre le conflit, les tâches $VALISES$ et $DEPART$ n'ont donc plus d'arc les reliant. Les relations de précédence induites doivent être explicitement données par l'utilisateur, dans notre cas l'utilisateur devrait rajouter la relation de précédence reliant les tâches $VALISES$ et $DEPART$.

Dans un problème $P =< T, S, R, L >$ d'ordonnancement, il y a 4 types de contraintes : les durées, les dates de livraison, les dates de disponibilité et les contraintes de précédence. Pour résoudre les conflits, nous indiquons un ensemble de contraintes à "enlever" *i.e.* une solution. En fonction des préférences de l'utilisateur les solutions respectent certains critères définis précédemment. Toutefois il arrive que dans la pratique, l'utilisateur ne puisse pas "supprimer" une contrainte, mais soit juste capable de la "modifier", de la "relaxer" pour résoudre un conflit *i.e.* modifier une des constantes présentes dans la contrainte. De plus, comme le montre l'exemple 19 page précédente, supprimer des contraintes peut avoir des effets inattendus. c'est pour cela que nous ne nous intéresserons qu'à la modification de constantes dans les contraintes.

4.1 Modification de constantes dans un PERT

Relaxer une constante *i.e.* diminuer la valeur d'une durée d_i ou d'une date de livraison l_i ou d'une date de disponibilité r_i, d'un conflit peut résoudre le conflit, mais pas forcément. Car rappelons qu'un conflit est un chemin inconsistant qui vérifie l'inéquation suivante :

$$\sum_{k \in i, \ldots, i+l} d_k > l_{i+l} - r_i$$

Remarque 34 *La diminution d'une valeur d'une des constantes ne rend pas nécessairement le chemin consistant (i.e. inverse l'inégalité)*

Définition 37 (Relaxation) *Soit un problème d'ordonnancement $P =< T, S, R, L >$ une relaxation $\mathcal{R}elax$ est la composition finie d'au moins une des trois relaxations élémentaires suivantes :*
- $\mathcal{R}r : P \rightarrow P'$ *où seule une date de disponibilité r_i est diminuée dans P' par rapport à P, r_i prend sa nouvelle valeur dans $[0, r_i[$.*
- $\mathcal{R}l : P \rightarrow P'$ *où seule une date de livraison l_i est augmentée dans P' par rapport à P, l_i prend sa nouvelle valeur dans $]l_i, \infty]$.*

- $\mathcal{R}d$: $P \to P'$ *où seule une durée est diminuée dans* P' *par rapport à* P, d_i *prend sa nouvelle valeur dans* $[0, d_i[$.

Trouver une solution à un problème d'ordonnancement revient donc à trouver une relaxation $\mathcal{R}elax$ qui rend consistant le problème.

4.2 De nouveaux critères spécifiques au PERT

La définition de conflits sous la forme particulière (Propriété 1 page 11) d'une date de livraison, d'une date de disponibilité et d'un ensemble de durées nous fait penser à relâcher les contraintes suivant le rôle qu'elles jouent. Cela nous donne l'idée d'introduire de nouveaux critères.

4.2.1 Relaxer une durée

Diminuer la valeur d'une durée d_i affecte tous les conflits qui contiennent d_i. Il est possible que cette diminution résolve tous les conflits, comme elle peut n'en résoudre qu'une partie ou même aucun. Il faut pour le savoir changer la durée d_i en sa nouvelle valeur et recalculer les conflits. Remarquons que l'on peut se contenter de ne recalculer que les conflits contenant d_i. Partant de cette constatation on va compter le nombre d'inéquations utilisant d_i dans une solution, pour pouvoir définir de nouveaux critères.

Les durées apparaissent dans les contraintes de précédence S[1] et les contraintes de livraison L.

Définition 38 (Nbdurée) *Soit un problème d'ordonnancement* P, $\mathcal{C} = \{c_i, i = 1, \ldots, n\}$ *l'ensemble des conflits de* P *et* Sol *une solution, on définit : Nbdurée(Sol)=* $|Sol \cap S| + |Sol \cap L|$

Remarque 35 *Nous pourrions définir un autre critère qui ne tiendrait compte que de la présence des durées sur les précédences.*

Pmaxd On cherche la solution qui contiendra le plus d'inéquations contenant une d_i (précédence et jalons), ainsi on touchera le moins possible aux contraintes extérieures (dates de livraison pour les clients et dates de disponibilité pour les fournisseurs).

Définition 39 (Pmaxd) *Soit* S_1, S_2 *deux solutions d'un problème de conflits* $\mathcal{C} = \{c_i, i = 1..n\}$.
$S_1 \leq_{Pmaxd} S_2$ *si et seulement si Nbdurée(*S_1*)* $\leq$ *Nbdurée(*S_2*).*

Pmind On cherche la solution qui contiendra le moins de d_i : on veut éviter de modifier les contraintes liées aux durées des tâches.

Définition 40 (Pmind) *Soit* S_1, S_2 *deux solutions d'un problème de conflits* $\mathcal{C} = \{c_i, i = 1..n\}$.
$S_1 \leq_{Pmind} S_2$ *si et seulement si Nbdurée(*S_1*)* $\geq$ *Nbdurée(*S_2*).*

4.2.2 Relaxer une date de livraison

Augmenter la valeur d'une date de livraison l_i affecte tous les conflits qui contiennent l_i *i.e.* qui finissent par l_i. Il est possible que cette diminution résolve tous les conflits, comme elle peut n'en résoudre qu'une partie ou même aucun. Il faut pour le savoir changer la valeur de l_i et recalculer les conflits. Remarquons que l'on peut se contenter de ne recalculer que les conflits finissant par l_i. Partant de cette constatation on va compter le nombre d'inéquations utilisant l_i dans une solution.

Définition 41 (NbLivraison) *Soit un problème d'ordonnancement* P, $\mathcal{C} = \{c_i, i = 1, \ldots, n\}$ *l'ensemble des conflits de* P *et* Sol *une solution on définit : NbLivraison(Sol)=* $|Sol \cap L|$

Pmaxl On cherche la solution qui contiendra le plus de l_i : on demande des efforts aux clients.

Définition 42 (Pmaxl) *Soit* S_1, S_2 *deux solutions d'un problème de conflits* $\mathcal{C} = \{c_i, i = 1..n\}$.
$S_1 \leq_{Pmaxl} S_2$ *si et seulement si NbLivraison(*S_1*)* $\leq$ *NbLivraison(*S_2*).*

[1] Lorsqu'une solution S peut se confondre avec S l'ensemble des contraintes de précédence d'un problème d'ordonnancement, on notera la solution Sol pour lever toute ambiguïté.

Pminl On cherche la solution qui contiendra le moins de l_i : on touchera le moins possible aux dates de livraison imposées par le client.

Définition 43 (Pminl) *Soit S_1, S_2 deux solutions d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$.*
$S_1 \leq_{Pminl} S_2$ si et seulement si NbLivraison(S_1) $\geq$ NbLivraison(S_2).

4.2.3 Relaxer une date de disponibilité

Diminuer la valeur d'une date de disponibilité r_i affecte tous les conflits qui contiennent r_i *i.e.* qui commencent par r_i. Il est possible que cette diminution ne résolve aucun, un, deux, ... tous ces conflits. Il faut pour le savoir changer la valeur de r_i et recalculer les conflits. Remarquons que l'on peut se contenter de ne recalculer que les conflits débutant par r_i. Partant de cette constatation on va compter le nombre d'inéquations utilisant r_i dans une solution.

Définition 44 (Nbdisponibilité) *Soit un problème d'ordonnancement P, $\mathcal{C} = \{c_i, i = 1, \dots, n\}$ l'ensemble des conflits de P et Sol une solution on définit : Nbdisponibilité(Sol)=$|Sol \cap R|$*

Pmaxr On cherche la solution qui contiendra le plus de r_i : on demande des efforts aux fournisseurs.

Définition 45 (Pmaxr) *Soit S_1, S_2 deux solutions d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$.*
$S_1 \leq_{Pmaxr} S_2$ si et seulement si Nbdisponibilité(S_1) $\leq$ Nbdisponibilité(S_2).

Pminr On cherche la solution qui contiendra le moins de r_i : on demande le moins possible d'efforts au fournisseurs.

Définition 46 (Pminr) *Soit S_1, S_2 deux solutions d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$.*
$S_1 \leq_{Pminr} S_2$ si et seulement si Nbdisponibilité(S_1) $\geq$ Nbdisponibilité(S_2).

4.2.4 Bilan

Nous utiliserons ces critères pour améliorer les critères du chapitre 2. Par exemple, il serait possible de chercher parmi les solutions parcimonieuses, celles optimales pour $\leq_{Pminr}$. Toutes les combinaisons possibles et imaginables entre ces critères et les différents critères du chapitre 2 peuvent être envisagées. Ceci permettra une recherche plus "fine" de la solution.

Nous cherchons maintenant une méthode de recherche de solutions, basée sur la relaxation de contraintes et la méthode de construction des conflits, prenant ainsi en compte la nature de notre problème.

4.3 Recherche de solution par la "méthode" de construction des conflits

Dans le chapitre 1, nous avons décrit comment construire tous les conflits d'un problème d'ordonnancement. Nous allons nous inspirer de cette méthode ainsi que de la technique employée par Castell et Leberre, pour cette fois résoudre les conflits, par modifications de constantes. Au lieu de partir de l'ensemble des conflits, nous allons trouver une solution au fur et à mesure que nous construisons les conflits. A chaque fois que nous rencontrons une nouvelle tâche qui débute un chemin inconsistant soit elle est composante de la solution soit elle ne l'est pas. En guidant ainsi l'algorithme de type "Davis et Putnam" nous développons un arbre de recherche de solutions sans avoir besoin de calculer les conflits.

Nous avons choisi ici de n'agir que sur les durées des tâches à effectuées, plus précisément les constantes de durée associées à la tâche sélectionnée deviendront nulles. Dans certaines situations, rendre toutes les durées nulles peut ne pas être suffisant pour résoudre tous les conflits. Nous nous autorisons alors à modifier les dates de disponibilité (on leur affectera une valeur négative arbitrairement grande par rapport à la durée totale du projet). On peut envisager que l'utilisateur nous donne pour chaque constante de chaque conflit, en plus de la couleur de la contrainte, un pourcentage indiquant la modération qu'il est paraît à effectuer. Au lieu de passer arbitrairement à zéro la valeur de la durée nous diminuerions sa valeur du pourcentage indiqué. Le nombre de solutions faisant recours à la modification des dates de disponibilité augmentera, s'il on considère des pourcentage au lieu d'une diminution radicale.

Nous adaptons le principe de Branch-and-Bound à la résolution de conflits dans l'algorithme 8 pour trouver les solutions suivant nos critères. Pratiquement, nous avons rajouter les lignes (1), (2), (3), (4), (5), (6) et (7) par rapport à l'algorithme de calcul de toutes les solutions.

Algorithme 8: Initialisation et lancement de la recherche de solution

% Initialisation de la variable globale $\mathcal{S}$ représentant l'ensemble des solutions %
$\mathcal{S} = \{\}$;
%Initialisation de la variable globale $NBSOL$ qui compte le nombre de solutions trouvées%
$NBSOL = 0$;
Calcul des dates au plus tôt de g;
Calcul des dates au plus tard de g;
Initialisation-Borne($Borne$**);**(1)
n = nombre de nœuds de g;
si g *inconsistant* **alors** `DP-Sol-ARR`(`TacheSuivante`(n,g,{},{}),$nbmaxsol$,g,{},{});

La procédure `TâcheSuivante` indique quelle sera la prochaine tâche à regarder, c'est à dire la prochaine tâche achevant un chemin inconsistant dans l'état courant.

Procedure `TâcheSuivante`(x_i : *int, graphe g,in : liste, out :liste*)

% On détermine la prochaine tâche à examiner %
début
  $trouver = false$;
  x_k= la dernière tâche de g;
  tant que $trouver = false$ *et* $x_k \neq \alpha^2$ **faire**
    si $k \notin out$ *et* $k \notin in$ *et* $\underline{t}_k > \overline{t}_k$ **alors**
      $trouver = true$;
    sinon
      $x_k = x_{k-1}$;
  si $x_k = \alpha$ **alors**
    return (x_i);
  sinon
    return (x_k);
fin

Exemple 20 *Soit l'ensemble de conflits $\mathcal{C} = \{\{a, b\}, \{b, c, \}\}$. Nous développons deux des différents arbres de recherches possibles. Les branches gauches des arbres de recherches sont les branches “IN” (on prend la formule dans la solution) et les branches droites les branches “OUT” (on ne prend pas la formule dans la solution). Nous entourons les solutions.*

Procedure `DP-Sol-ARR`(x_i : *tache*,*nbmaxsol : int, g : graphe, in : liste, out liste*)

% On construit les branches IN et OUT l'arbre de recherche avec la tache x_i %

début

    %Le nombre maximum de solutions est atteint : on quitte l'algorithme%
    si ($NBSOL < nbmaxsol$) **alors**
        si *g est consistant* **alors**
            Ajouter *in* à S;
            Mise-a-jour-B($Borne$**);**(2)
            $NBSOL = NBSOL + 1$;
            %Le graphe ne contient plus de conflits donc cette branche est totalement explorée passons à la suivante %
        sinon
            Si non(Test-coupure(S,$Borne$,$Bornecourante$**));**(3)
            si $x_i \notin out$ *et* $x_i \notin in$ **alors**
                `IN`(x_i,*nbmaxsol*,*g*,*in*,*out*);
                `OUT`(x_i,*nbmaxsol*,*g*,*in*,*out*);
            sinon
                si $r[x_i] > plustard[x_i]$ **alors**
                    `ultime`(x_i,*nbmaxsol*,*g*,*in*,*out*);
    % Sinon on coupe la branche%

fin

Procedure `ultime`(x_i : *tache, nbmaxsol : int, g : graphe, in : liste, out liste*)

% Dernière chance de résoudre le conflit en modifiant une date de disponibilité %

début

    Ajouter-liste($r[x_i], in$);
    % On relâche $r[x_i]$%
    Modifier la disponibilité de x_i dans g à -1000;
    Mise-a-jour-BC ($Bornecourante$,x,$\mathcal{C}$, S)**);**(4)
    Calcul des dates au plus tôt de g ;
    Calcul des dates au plus tard de g;
    % On passe au conflit suivant%
    `DP-Sol-ARR`(`TâcheSuivante`(x_i,*g*,*in*,*out*),*nbmaxsol*,*g*,*in*,*out*);
    Rétablir la disponibilité de x_i dans g;
    Mise-a-jour-BC ($Bornecourante$,x,$\mathcal{C}$, S)**);**(5)
    Calcul des dates au plus tôt de g;
    Calcul des dates au plus tard de g ;
    % Enlever le dernier élément ajouté à *in* %
    Pop(*in*);

fin

Procedure `IN`(x_i : *int,nbmaxsol : int ,g : Graphe, in : liste, out :liste*)

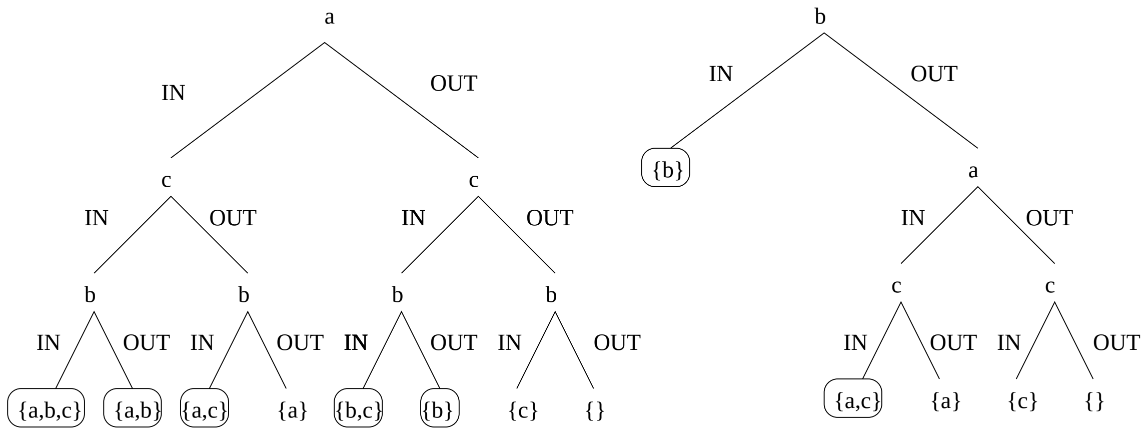
% On met la tâche x_i dans in, on effectue les modification nécessaire et on relance la construction de l'arbre de recherche%
début
- Ajouter-liste($d[x_i], in$);
- Modifier la durée de x_i à 0 dans g;
- **Mise-a-jour-BC (**$Bornecourante$**,**$x, \mathcal{C}, S$**));**(6)
- Calcul des dates au plus tôt de g ;
- Calcul des dates au plus tard de g;
- `DP-Sol-ARR`(`TâcheSuivante`(x_i,*g*,*in*,*out*),*nbmaxsol*,*g*,*in*,*out*);
- Rétablir la durée de x_i dans g;
- **Mise-a-jour-BC (**$Bornecourante$**,**$x, \mathcal{C}, S$**));**(7)
- Calcul des dates au plus tôt de g ;
- Calcul des dates au plus tard de g;
- % Enlever le dernier élément ajouté à *in* %
- Pop(in);

fin

Procedure `OUT`(x_i : *int, nbmaxsol : int ,g : Graphe, in : liste, out :liste*)

% On met la tâche x_i dans *out* et on relance la construction de l'arbre de recherche%
début
- Ajouter($d[x_i], out$);
- `DP-Sol-ARR`(`TâcheSuivante`(x_i,*g*,*in*,*out*),*nbmaxsol*,*g*,*in*,*out*);
- pop(out);

fin



Remarque 36 *Dans l'exemple 20 page 41, dans le "meilleur" des cas, on choisit* $\{b\}$ *en premier, on trouve 2 solutions* $\{b\}$ *et* $\{a, c\}$ *et on atteint 4 feuilles. Dans le second cas, "le pire", on prend* $\{a\}$ *puis* $\{c\}$ *et* $\{b\}$*, on trouve 5 solutions* $\{a, b, c\}$,$\{a, b\}$,$\{a, c\}$,$\{b, c\}$ *et* $\{b\}$*, et on atteint 8 feuilles. La fonction* `TâcheSuivante` *effectue le choix de l'élément suivant à "tester". Elle joue un rôle crucial en fonction du critère à respecter.*

4.4 Résultats des expérimentations

L'algorithme 8 page 41 est implémenté en C, sur une station SUN microsystems Ultra 5, avec 128 Mo de RAM, 332 Mo de swap et le système d'exploitation SUN-OS release 5.7. Le programme est testé sur des exemples de problèmes d'ordonnancement issus de la bibliothèque de problèmes "PSPLIB", créée par Kolisch et Sprecher dans [KS97]. Cette Bibliothèque contient de nombreux problèmes d'ordonnancement de tâches classés en 4 catégories : K30, K60, K90 et K120. Cette classification regroupe les problèmes en fonction du nombre de tâches des problèmes, K30 ne contient que des problèmes à 30 tâches, K60 ne contient que des problèmes à 60 tâches ... La méthode de génération de ces problèmes est détaillée dans [KSD95].

Pour effectuer les tests, nous prenons un ensemble de 24 problèmes de la bibliothèque K30. Les tâches de ces problèmes ne possèdent pas de date de livraison ni de date de disponibilité. Nous affectons donc à toutes les tâches une date de disponibilité nulle et une date de livraison égale à la durée maximale du projet. Pour chaque problème, nous diminuons alors progressivement la date de livraison de sa dernière tâche. Ainsi nous augmentons progressivement le nombre de conflits et la taille des conflits. A chaque diminution, nous recherchons à nouveau les solutions.

Nous regardons alors le temps de résolution pour le calcul de toutes les solutions et des solutions parcimonieuses par rapport à deux heuristiques choisies (dans la fonction `TâcheSuivante`) :

Heuristique 1 : nous prenons la première tâche conflictuelle grâce à un parcours du graphe en propagation arrière, *i.e.* on prend la tâche conflictuelle ayant le plus grand numéro (heuristique très naïve).

Heuristique 2 : on recherche la prochaine tâche conflictuelle sur les prédécesseurs de la tâche courante *i.e.* on reste sur le “conflit courant”, s’il n’existe pas de tâche conflictuelle dans les prédécesseurs de la tâche courante, on appliquera l’heuristique 1.

Remarque 37 *L’heuristique 1 nous fait penser à une sorte de recherche en largeur d’abord alors que la seconde correspondrait plutôt à une recherche en profondeur d’abord dans l’exploration du graphe des conflits.*

4.4.1 Comparaison entre H1 et H2

Nous comparons d’abord les deux heuristiques pour le calcul de toutes les solutions. On n’utilise pas les lignes (1), (2), (3), (4), (5), (6) et (7) du “Branch-and-Bound” de l’algorithme. Ensuite nous regarderons les différents résultats pour le calcul des solutions parcimonieuses en utilisant cette fois si la technique du “Branch-and-Bound”.

4.4.1.1 Calcul de toutes les solutions

Pour obtenir toutes les solutions l’heuristique 2 met moins de temps que l’heuristique 1 (cf graphique 4.1). Elle calcule moins de solutions en construisant un arbre de recherche plus petit.

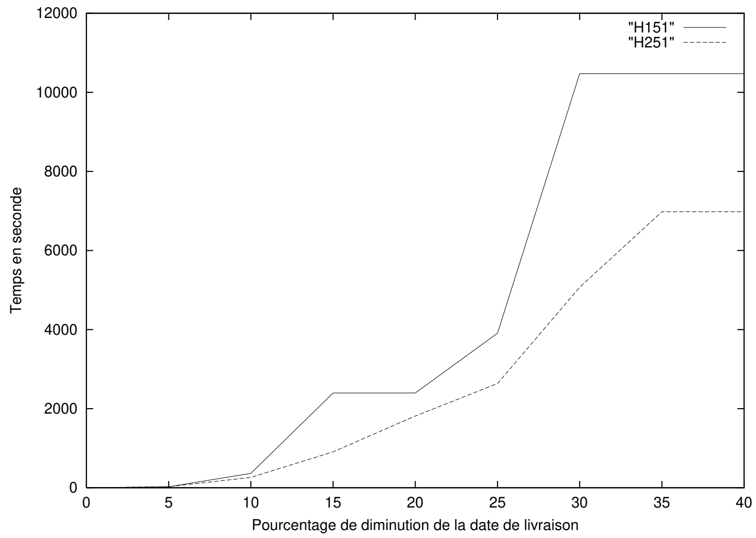


FIG. 4.1: Comparaison entre H1 et H2 pour le calcul de “toutes” les solutions

4.4.1.2 Calcul des solutions parcimonieuses

Le graphique 4.2 page suivante montre que l’heuristique 2 reste meilleure que l’heuristique 1 pour le calcul des solutions parcimonieuses, mais, cette fois-ci, l’amélioration est beaucoup moins significative. On peut supposer que cela vient du fait que résoudre entièrement le premier conflit rencontré donnera une première solution de taille plus grande que celle fournie par l'heuristique 1. Ainsi le test de coupure opérera plus efficacement au début pour l'heuristique 1 que pour l'heuristique 2.

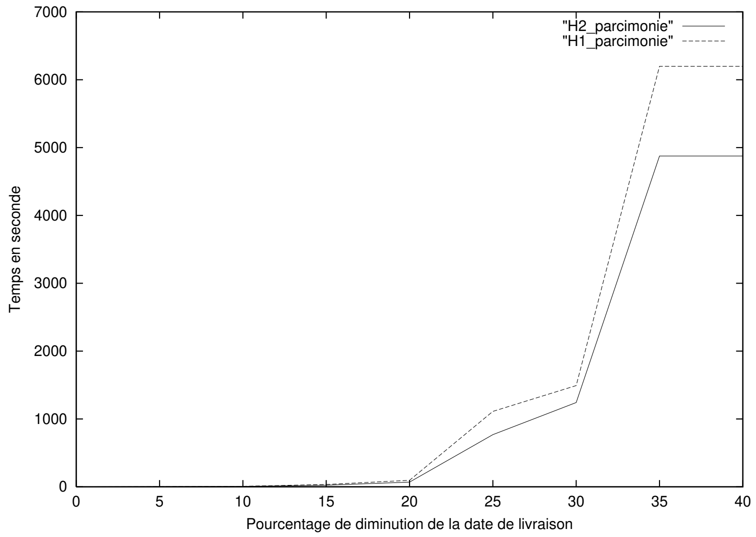


FIG. 4.2: Comparaison entre H1 et H2 pour le calcul des solutions parcimonieuses

Remarque 38 *Le choix de la tâche conflictuelle effectué dans* `TâcheSuivante` *est important, un choix judicieux diminue l'arbre de recherche et donc améliore considérablement la recherche de solution suivant un critère choisi, comme le montre l'exemple 20 page 41.*

4.4.2 Résultats de faisabilité

Sur le graphique 4.3, nous observons les résultats associés à l'heuristique 2 pour le calcul de toutes les solutions et pour le calcul des solutions parcimonieuses.

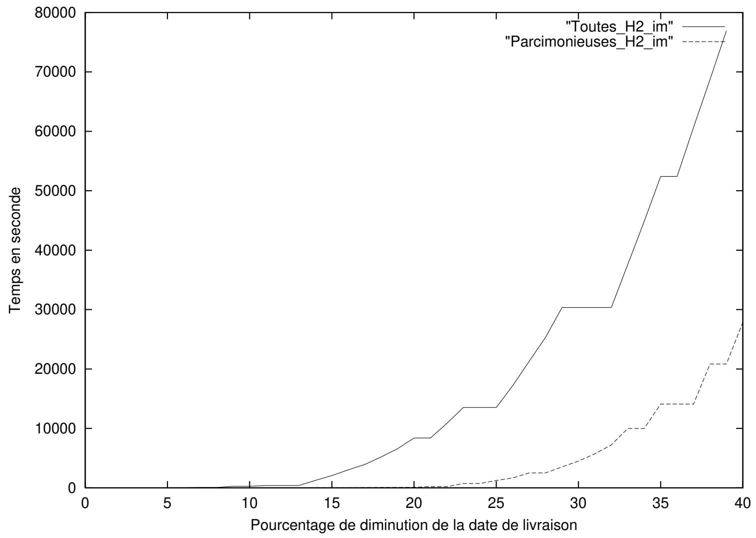


FIG. 4.3: Comparaison entre le calcul de toutes les solutions et le calcul des solutions parcimonieuses pour H2

On remarque que plus la diminution de la date de livraison de la dernière tâche est importante, plus le temps de résolution du problème augmente. Cette augmentation est même exponentielle pour le calcul de toutes les solutions et pour le calcul des solutions parcimonieuses. Cela est dû au fait, que plus la date de livraison de la dernière tâche est petite plus il y aura de conflits à résoudre et plus le nombre d'éléments dans les conflits sera grand, donc plus il y aura de possibilités différentes pour construire une solution. L'arbre de recherche sera donc plus grand.

4.5 Conclusion

Trouver une solution, c'est rendre consistant un problème d'ordonnancement inconsistant, en trouvant une "relaxation" (donc une modification de valeur) pour les constantes des contraintes conflictuelles. Nous constatons que plus le nombre de confits et plus les conflits contiennent d'éléments, plus le temps de résolution est élevé. Dans l'heuristique 2, pour choisir l'élément qui constituera le prochain nœud de l'arbre de recherche, nous nous sommes inspirés du principe mis en œuvre pour le calcul des conflits : "Tant qu'il existe des prédécesseurs conflictuels de la tâche courante, nous continuerons à résoudre ce conflit". Les différents tests effectués montrent que cette heuristique accélère le processus de recherche par rapport à l'heuristique 1 qui choisit la première tâche conflictuelle. Castell et Leberre ont proposé et étudié de nombreuses heuristiques spécifiques à leurs domaines, essayer de les transposer demeure un travail intéressant à faire.

Conclusion

Quand un problème d'ordonnancement de tâches ne possède pas de solution, nous pouvons calculer les conflits : ensembles de contraintes "incohérents" . Il est possible en modifiant ces ensembles de rendre le problème consistant. Nous avons donc cherché à effectuer ces modifications en accord avec les préférences de l'utilisateur. Pour cela nous avons adopté une approche locale *i.e.* nous avons demandé à l'utilisateur qu'il nous indique sur chaque conflits quelles étaient ces préférences. Nous avons étudié les propriétés des différents critères de préférences locales de résolution de conflit existant en logique propositionnelle, et nous avons adaptés ces critères aux problèmes d'ordonnancement. Nous avons également imaginé et étudié de nouveaux critères spécifiques basés sur une coloration locale des conflits, pour représenter les préférences de l'utilisateur. Nous avons alors défini la solution "parfaite", solution qui respecte le plus les préférences exprimées par l'utilisateur. Parmi tous les critères transitifs, nous avons sélectionné ceux qui permettent de calculer une solution la plus proche de la solution parfaite. Une fois les critères de construction des solutions établis, nous avons cherché à partir d'un problème d'ordonnancement, à calculer les solutions optimales pour ces critères.

En remarquant que pour modifier un problème d'ordonnancement inconsistant en un problème consistant, plusieurs possibilités étaient envisageables, nous avons opté pour celle qui consiste à "relaxer" les constantes présentes dans les contraintes sélectionnées (une "relaxation" est une diminution de la valeur de la constante associée à la contrainte choisie). Nous avons choisi plus précisément de passer à 0 les constantes de durée dans les contraintes choisies. Ensuite nous observons à nouveau les conflits du problème modifié. Trouver une solution consiste alors à trouver plusieurs relaxations successives pour obtenir un problème modifié consistant.

Nous nous sommes inspirés des techniques utilisées par Castell [Cas97] et Leberre [Leb00] pour construire un algorithme de recherche de solutions optimales d'un ensemble de conflits d'un problème d'ordonnancement, par cette méthode de relaxation. Ensuite, nous avons amélioré tous nos algorithmes par la technique dite de "Branch-and-Bound". Cette technique permet de ne pas développer certaines branches de l'arbre de recherche qui ne fourniront pas de solutions optimales pour le critère retenu.

Nous avons plus particulièrement implémenté cette méthode de recherche de solutions pour le calcul de toutes les solutions et pour le calcul des solutions parcimonieuses. Pour améliorer encore l'efficacité de cette méthode nous avons proposé deux heuristiques différentes pour le choix de l'élément courant, lors de la construction d'une solution. La première heuristique (très naïve) consiste à prendre la première tâche conflictuelle rencontrée en parcourant par propagation arrière le graphe, associé au problème d'ordonnancement. La seconde heuristique est une légère "amélioration" de la première, on sélectionne la tâche conflictuelle suivante parmi les prédécesseurs de la dernière tâche choisie. Cette heuristique est basée sur le même principe que le calcul des conflits *i.e.* on reste sur le même chemin inconsistant tant qu'il n'est pas résolu. L'implémentation de ces deux heuristiques montre que la seconde heuristique trouve les solutions parcimonieuses ainsi que toutes les solutions plus rapidement que la première.

Perspectives

Plusieurs points restent à développer :
- Envisager d'autres types de "relaxation" *i.e.* une diminution partielle de la valeur de la constante.
- Chercher des algorithmes pour les critères non-transitifs.

- Regarder si permuter IN et OUT ne peut pas améliorer la recherche de certaines solutions optimales.
- Tester l'heuristique 2 sur tous les critères évoqués pour regarder si elle apporte une amélioration dans tous les cas.
- Utiliser la technique "diviser pour régner" en divisant les conflits en conflits "indépendants" pour les résoudre séparément et donc paralléliser le travail, semble être une approche intéressante.
- Rechercher de nouvelles heuristiques :
  - Développer d'autres heuristiques qui prennent en compte les conflits pour la construction des solutions.
  - Regarder les heuristique de type FIS (Favoring Identifiyng a satisfiable solution) proposées par [Wan97], par exemple considère l'heuristique MOMS (Maximum Occurrences in clauses of Minimum Size) qui se transposerait à choisir l'élément qui apparaîtrait dans le plus de conflits de tailles minimales.
  - Regarder les heuristiques proposées par Castell ou Leberre et les adaptées à notre problème.
  - Envisager une approche évoquée par Leberre, développée par Korf dans [Kor96], qui est la technique de LDS (Limited Discrepancy Search) pour accélérer la recherche de solutions.

Annexe A

Notations

Nous récapitulons les différentes notations utilisées dans les différentes parties.

- S ensemble de contraintes de précédence, de la forme $t_j \geq t_i + d_i$, *i.e.* la tâche x_j doit être réalisée après la tâche x_i.
- R ensemble de contraintes de disponibilité, de la forme $t_i \geq r_i$, *i.e.* la tâche x_i ne peut commencer qu'après la date r_i.
- L ensemble de contraintes de livraison, de la forme $t_i + d_i \leq l_i$, *i.e.* la tâche x_i doit se finir avant la date l_i.
- $< T, S, R, L >$ représente un problème d'ordonnancement.
- Soit un graphe orienté G :
  - x_{i+1} est successeur de x_i si (x_i, x_{i+1}) est un arc.
  - x_i est prédécesseur de x_{i+1} si (x_i, x_{i+1}) est un arc.
- la date de début au plus tôt de la tâche t_i sera notée $\underline{t_i}$.
- la date de début au plus tard de la tâche t_i sera notée $\overline{t_i}$.
- les formules non-modifiables (les $r(c_i)$) en **gras**.
- les formules éventuellement modifiables (les $b(c_i)$) écrites normalement.
- les formules modifiables (les $v(c_i)$) <u>soulignées</u>.
- $\mathcal{R} = \cup_{j \in \{1..n\}} r(c_j)$ ensemble des formules non-modifiables de $\mathcal{C}$[1].
- $B = \cup_{j \in \{1..n\}} b(c_j)$ ensemble des formules éventuellement modifiables de $\mathcal{C}$.
- $V = \cup_{j \in \{1..n\}} v(c_j)$ ensemble des formules modifiables de $\mathcal{C}$.
- les solutions minimales sont <u>soulignées</u>.
- les étoiles représentent les solutions minimales préférées.
- S_1 et S_2 dénotant des solutions d'un problème de conflits $\mathcal{C} = \{c_i, i = 1..n\}$ et $\leq_x$ une relation d'ordre alors $S_1 \leq_x S_2$ signifie que S_2 est meilleure que S_1 selon "x".

[1]Attention à ne pas confondre avec R : l'ensemble des contraintes de disponibilités

Annexe B

Étude des différents critères sur les exemples du chapitre 2

Notation :
- les cases du tableau contiennent les solutions optimales au sens des critères indiqués sur les lignes, sur les différents exemples.
- S_{123} correspond à S_1, S_2, S_3
- Tous : signifie que toutes les solutions remplissent le critère.
- non-sens : il n'y a pas de préférence sur les conflits donc les critères portant sur les préférences n'ont pas de sens. Autre cas possible, on ne peut "départager" les solutions trouvées au sens du critère (problème pour les critères ne respectant par la transitivité).
- $\emptyset$: signifie que il n'y a pas de solution qui remplit le critère.

Critère	exemple 9 page 19	exemple 10 page 19	exemple 11 page 21
Minimal (def 16 page 17)	S_1, S_2, S_3	S_1, S_2, S_3, S_4	S_1, S_2, S_3
Min Forte (def 18 page 18)	$\emptyset$	S_2, S_3, S_4	$\emptyset$
Card (min)(def 17 page 18)	S_1, S_2, S_3	S_1	S_1, S_2, S_3
V1f (max) (def 29 page 25)	non-sens	non-sens	S_1, S_3
B1f (min) (def 28 page 24)	non-sens	non-sens	S_1, S_3
Pref (def 22 page 21)	non-sens	non-sens	$\emptyset$
Pref-F (def 27 page 23)	non-sens	non-sens	$\emptyset$
Pmaxmin(def 20 page 20)	S_1, S_2, S_3	S_2, S_3, S_4	S_1, S_2, S_3
Pvert(max)(def 31 page 25)	non-sens	non-sens	S_1, S_3
Pbleu(min)(def 33 page 26)	non-sens	non-sens	S_1, S_3
eli (def 23 page 21)	non-sens	non-sens	$\emptyset$
eli-F (def 24 page 22)	non-sens	non-sens	$\emptyset$

Critère	exemple 12 page 21	exemple 13 page 21	exemple 14 page 23
Minimal (def 16 page 17)	S_1, S_2, S_3, S_4	Tous	Tous
Min Forte (def 18 page 18)	S_1	$S_{12356789}, S_{10}$	$S_{134689}, S_{10}, S_{11}$
Card (min)(def 17 page 18)	S_1	S_1	S_1
V1f (max) (def 29 page 25)	S_2	S_{2347}	S_{2356}, S_{10}
B1f (min) (def 28 page 24)	S_1, S_2	S_{1234}	S_6
Pref (def 22 page 21)	S_1, S_2	S_{1234}	S_{123}, S_5, S_6
Pref-F (def 27 page 23)	$\emptyset$	$\emptyset$	S_6
Pmaxmin(def 20 page 20)	S_1	$S_{12356789}, S_{10}$	$S_{134689}, S_{10}, S_{11}$
Pvert(max)(def 31 page 25)	S_2	S_4	S_5, S_6
Pbleu(min)(def 33 page 26)	S_2	S_{234}	S_6
eli (def 23 page 21)	$S_1 S_2$	non-sens	S_1, S_3, S_6
eli-F (def 24 page 22)	S_2	non-sens	S_1, S_3, S_6

Critère	exemple 15 page 23	exemple 16 page 24	exemple 18 page 26
Minimal (def 16 page 17)	S_4, S_5	S_1, S_2, S_8	S_1
Min Forte (def 18 page 18)	S_4, S_5	S_1, S_2, S_8	S_4
Card (min)(def 17 page 18)	S_5	S_1, S_2	S_1
V1f (max) (def 29 page 25)	S_1, S_3	S_6	S_4
B1f (min) (def 28 page 24)	S_3, S_4, S_5	S_{12468}	S_{1234}
Pref (def 22 page 21)	S_5	S_2	S_1
Pref-F (def 27 page 23)	$\emptyset$	$\emptyset$	$\emptyset$
Pmaxmin(def 20 page 20)	S_4, S_5	S_1, S_2, S_8	S_1
Pvert(max)(def 31 page 25)	S_4, S_5	S_6	S_1, S_2
Pbleu(min)(def 33 page 26)	S_4, S_5	S_2, S_6, S_8	S_4
eli (def 23 page 21)	non-sens	non-sens	non-sens
eli-F (def 24 page 22)	non-sens	non-sens	non-sens

TAB. B.1: Récapitulatif des critères sur les exemples.

Annexe C

Rappel sur les graphes orientés

Nous donnons une série de définitions et de propriétés sur les graphes que l'on retrouve dans [Ber70, Die98, Kau64]. Ces notions seront utilisées dans ce document pour formaliser les problèmes d'ordonnancement sous forme de graphe.

Définition 47 (Graphe orienté) *Un graphe orienté G est un couple $G = (X, U)$, où X est un ensemble (fini) appelé ensemble de sommets et U une partie du produit cartésien X^2, appelé ensemble des arcs de G.*

Définition 48 (Graphe valué) *Un graphe dont chaque arc (x_i, x_j) possède une valeur $d_{i,j}$ sera dit valué.*

Définition 49 (Chemin) *Un chemin du graphe orienté $G = (X, U)$ est une séquence $(x_1, x_2, \dots, x_k), k \geq 2$ de sommets de X tel que :*

- Tout couple de sommets consécutifs de la séquence est un arc de U : $\forall i = 1, \dots, k-1, (x_i, x_{i+1}) \in U$

Un chemin reliant le sommet x_i au sommet x_j sera symbolisé par l'ensemble des sommets permettant de relier x_i à x_j, i.e. $(x_i, \dots, x_j)$.

Définition 50 (Chemin élémentaire) *Un chemin dont tous les sommets sont différents est un chemin élémentaire.*

Définition 51 (Circuit) *Soit G un graphe orienté, un circuit est un chemin partant de x_i et arrivant en x_i, noté $(x_i, \dots, x_{i+l}, x_i)$.*

Définition 52 (Coût d'un chemin) *Soit G un graphe valué orienté et un chemin $c = (x_i, \dots, x_{i+l})$, le coût de c, noté $L(c)$, est défini par :*

$$L(c) = \sum_{k \in \{i, \dots, i+l-1\}} d_{k,k+1}$$

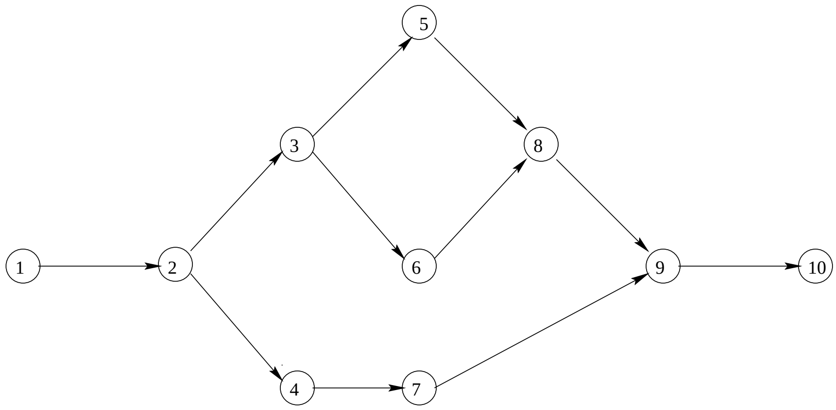
i.e. la somme des valeurs affectées aux arcs composant le chemin.

Définition 53 (Successeur/Prédécesseur) *Soit un graphe orienté G :*
- *x_{i+1} est successeur de x_i si (x_i, x_{i+1}) est un arc.*
- *x_i est prédécesseur de x_{i+1} si (x_i, x_{i+1}) est un arc.*

Notation :
- l'ensemble des successeurs de x_i sera noté $succ(x_i)$.
- l'ensemble des prédécesseurs de x_i sera noté $pred(x_i)$.

Définition 54 (Puits/Source) *Soit G un graphe orienté.*
- *un sommet sans successeur est appelé puits de G.*
- *un sommet sans prédécesseur est appelé source de G.*

Exemple 21 *Le graphe ci-dessous possède une source (le sommet 1), un puits (le sommet 10) et 3 chemins allant de 1 à 10 :*
- $(1, 2, 3, 5, 8, 9, 10)$
- $(1, 2, 3, 6, 8, 9, 10)$
- $(1, 2, 4, 7, 9, 10)$



Propriété 11 *Dans un graphe sans circuit, nous pouvons numéroter, grâce à l'algorithme 9 les sommets de telle sorte que si le sommet x_i précède le sommet x_j dans le graphe alors le numéro associé à x_i est strictement inférieur à celui associé à x_j.*

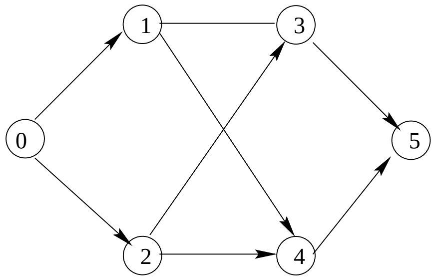
Algorithme 9: Algorithme de numérotation des nœuds d'un graphe orienté.

$G(X, U)$ un graphe orienté sans circuit
% **Nous appelons N_i les niveaux du graphe.** %
début
  $i \leftarrow 0$;
  $cpt \leftarrow 1$;
  tant que $X \neq \emptyset$ **faire**
    $N_i \leftarrow \{\text{sources de } G\}$;
    % Supprimer de G les nœuds de N_i%
    $X \leftarrow X \setminus N_i$;
    %On enlève tous les arcs partant des nœuds de N_i dans U %
    supprimer$(U, N_i\ )$;
    % Numérotons les nœuds de N_i %
    pour *tous les éléments* $x \in N_i$ **faire**
      Affecter le numéro *cpt* au nœud x;
      $cpt \leftarrow cpt + 1$;
    $i \leftarrow i + 1$;
fin

Définition 55 (Graphe série-parallèle) *[Gal00] Un graphe est série-parallèle si et seulement s'il est défini récursivement de la manière suivante :*

- *Un graphe ne contenant que deux sommets (une source et un puits) reliés entre eux par un arc est un graphe série-parallèle.*
- *Soit G_1 et G_2 deux graphes série-parallèle alors le graphe résultant d'une des deux opérations suivantes est un graphe série-parallèle :*
  Parallèle : Identifions la source de G_1 avec celle de G_2 et le puits de G_1 avec celui de G_2
  Série : Identifions le puits de G_1 avec la source de G_2.

Exemple 22 *Le graphe ci-dessous n'est pas série parallèle, alors que le graphe de l'exemple 21 l'est.*



Propriété 12 *Un graphe série-parallèle possède exactement une seule source et un seul puits.*

La réciproque est fausse, comme le montre l'exemple 22

Preuve : Par induction sur les différentes opérations de construction d'un graphe série-parallèle.

- Un graphe ne contenant que deux sommets (une source et un puits) reliés entre eux par un arc, possède exactement une source et un seul puits.
- Soit G_1 et G_2 deux graphes série-parallèle possédant exactement une source et un seul puits.
  Parallèle : Identifions la source de G_1 avec celle de G_2 et le puits de G_1 avec celui de G_2, le graphe résultant possède exactement une source et un seul puits.
  Série : Identifions le puits de G_1 avec la source de G_2, le graphe résultant possède exactement une source et un seul puits..

Nous avons bien montré qu'un graphe série-parallèle possède exactement une seule source et un seul puits.
□

Index

T

Bibliographie

[Ber70] C. Berge. *Graphes et hypergraphes.* Dunod, 1970. p8 et p158, 1970.

[BKSO96] P. Brucker, S. Knust, A. Schoo, et O.Thiele. A branch and bound algorithm for the resource-constrained project scheduling problem. *European Journal of Operational Research 107, 272-288*, 1996.

[Cas96] T. Castell. Computation of prime implicates and prime implicants by a variant of the davis and putnam procedure. Dans *Proceedings of the Eighth International Conference on Tools with Artificial Intelligence*, pages 428–429. IEEE, 1996.

[Cas97] T. Castell. *Consistance et déduction en logique propositionnelle.* Thèse, Université Paul Sabatier, Toulouse, France, 1997.

[CLSM00] C. Cayrol, M C. Lagasquie-Schiex, et J. Mengin. Une approche méthodologique de la résolution de conflits. Dans *Proc. of the 12th RFIA*, pages 215–224 (vol. 1), Paris, France, 2000.

[Die98] R. Diestel. *Graph Theory.* Springer, 1998.

[DP60] Davis et Putnam. A computing procedure for quantification theory. *JACM*, 7:201–215, 1960.

[EGJ02] A. Elkhyari, C. Gueéret, et N. Jussien. Conflict-based repair techniques for solving dynamic scheduling problems. Dans *Proc. (CP 2002)*, pages 702–707, Ithaca, NY, USA, September 2002. Morgan Kaufmann.

[EL99] P. Esquirol et P. Lopez. *L'Ordonnancement.* Economica 1999, Collection Gestion, 1999.

[FGT02] H. Fargier, V. Galvagnon, et C. Thierry. Explaining the inconsistency of project scheduling problems. Version préliminaire, Université Paul Sabatier, IRIT, juin 2002., 2002.

[Gal00] V. Galvagnon. *Aide à la décision en gestion multi-projet distribuée : Approche locale pour la planification à moyen terme.* Thèse, Université Paul Sabatier, Toulouse, France, 2000.

[GJ79] M. R. Garey et D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-completeness.* W.H. Freeman and Company, New York, 1979.

[GM85] M. Gondran et M. Minoux. *Graphes et algorithmes.* Eyrolles, 1985. p16 et p 54-56.

[GNZ86] J. Grabowski, E. Nowicki, et S. Zdrzalka. A block approach for single machine scheduling with release dates and due dates. *European Journal of Operational Research 26, 278-285*, 1986.

[GSW89] R. Greiner, B. A. Smith, et R. W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41:79–88, 1989.

[Kau64] A. Kaufmann. *Méthodes et modèles de la recherche opérationnelle.* Dunod, 1964. Tome 2, p56 et p350.

[Kor96] R. E. Korf. Improved limited discrepancy search. Dans *AAAI IAAI, Vol. 1*, pages 286–291, 1996.

[KP] R. Kolisch et R. Padman. An integrated survey of project scheduling.

[KS97] R. Kolisch et A. Sprecher. Psplib — a project scheduling problem library, 1997.

[KSD95] R. Kolisch, A. Sprecher, et A. Drexl. Characterization and generation of a general class of resource-constrained project scheduling problems, 1995.

[Leb00] D. Leberre. *Autour de SAT : le calcul d'impliquants P-restreints, algorithmes et applications.* Thèse, Université Paul Sabatier, Toulouse, France, 2000.

[Lev92] F. Levy. Reason maintenance systems and default theories. Pré-publication 92-2, LIPN, Université Paris Nord, 1992.

[LW66] E. L. Lawler et D. E. Wood. *Branch-and-bound methods: A survey*. Operations Research, 1966.

[MRCF59] D.G. Malcolm, J.H. Roseboom, C.E. Clark, et W. Fazar. Application of technique for research and development program evaluation. *Operations Research*, 7:646–669, 1959.

[Pap94] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.

[Rei87] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[Roy60] B. Roy. Contribution de la théorie des graphes à l'étude des problèmes d'ordonnancement. Dans *Actes du congrès*, pages 428–429. IFORS, 1960.

[SFV95] T. Schiex, H. Fargier, et G. Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. Dans *Proc. of the* 14^{th} *IJCAI*, pages 631–637, Montréal, Canada, 1995. Morgan Kaufmann.

[Wan97] J. Wang. Branching rules for propositionnal satisfiability test. Dans *Satidiability Problem : Theory and applications*, pages 351–364. American Mathematical Society, 1997.