UNIVERSITÉ DE NANTES

POLYTECH NANTES

# Mini-projet d'Intelligence Artificielle

## Planification des emplois du temps à Polytech

| | |
|---|---|
| **Encadrant** | Hoël LE CAPITAINE |
| **Date** | 13 janvier 2017 |

Nicolas GAILLARD
Jimmy LEBLANC

ID4

# Table des matières :

# Partie 2 : Implémentation de la solution

## 1) Introduction

Suite aux indications qui nous ont été apportées après le premier rendu, la conception a été modifiée. En effet, certaines parties de notre modélisation ne permettaient pas une bonne implémentation notamment la notion de temps qui empêchaient la mise en place d'un ordonnancement des matières. De nombreuses contraintes ont donc été repensées pour s'adapter à cette nouvelle conception.

A l'heure actuelle, l'algorithme traite "dynamiquement" une centaine de séances. Elles correspondent simplement au premier mois de l'année scolaire (soit le septembre 2016).

Le programme Prolog ainsi que le jeu de données sont joints dans la même archive que ce rapport. Une fois le programme "`poly_planification.pl`" consulté, il suffit d'entrer le prédicat "`faire_planification().`" afin de générer une solution.

## 2) Traduction de l'algorithme de résolution

Notre solution :

```
planifier([],Solution):- ecrireSolution(Solution).
planifier(ListeSeances,Solution):-

	% Choix non déterministe :
	% -------------------------
	member(Seance, ListeSeances),
	date(Jour, Mois),
	plage(Plage,_,_),
	salle(Salle),

	% Vérification des contraintes :
	% ------------------------------

	% Celles qui n'ont pas besoin de parcourir la solution :
	contrainteCM(Seance,Plage),
	contrainteJeudi(Jour,Plage),
	contrainteUsage(Seance,Salle),
	contrainteTailleSalle(Seance,Salle),
	contrainteSalleLibre(Plage,Jour,Mois,Salle,Solution),

	% Celles qui ont besoin de parcourir la solution :
	verificationEs(Seance,Salle,Plage,Jour,Mois,Solution),

	% Ajout de la plannification dans le résultat :
	% ---------------------------------------------
	append([ [Seance, Salle, Plage, Jour, Mois] ], Solution, Result),
	delete(ListeSeances , Seance, ListeTronquee),

	% Appel récursif de la solution :
	% -------------------------------
	planifier(ListeTronquee, Result).
```

Stratégie de l'algorithme :

Prédicat principal : `planifier(+ListeSeances, -Solution)`

Notre algorithme détermine une liste d'événements. Un événement est simplement un quintuplet composé d'une séance, une salle, une plage, un jour et un mois.

Il commence par réaliser la totalité des choix non déterministe. Il récupère premièrement une séance à planifier (pour des raisons de performances) et génère une date valide. On appelle date l'association d'un jour, d'un mois et d'une plage. La salle est ensuite déterminée.

Les contraintes post-conditions ont directement été intégrées à l'algorithme (c'est-à-dire pendant le traitement des séances) et ne se font donc plus une fois qu'une solution est déterminée. Cela a pour conséquence une détection anticipée des impasses et donc une amélioration significative des performances du programme.

Les contraintes ne nécessitant pas un parcours des événements déjà planifiés sont ensuite évaluées :

- un cours magistral n'est pas sur certains créneaux
- aucun cours ne peut être planifié les jeudis après midi
- la salle est adaptée au cours donné
- la salle est assez grande pour accueillir la totalité de groupes assistant au cours
- la salle est libre à la date choisie précédemment

Une fois ces contraintes passées, un parcours des évènements déjà planifiés est effectué afin de "valider" d'autres contraintes (qui se fait via le prédicat "`verificationEs`") :

- l'enseignant est disponible à la date donnée
- deux groupes incompatibles ne peuvent avoir cours en même temps
- respect de l'ordre des cours : un cours de projet d'IA ne peut avoir lieu avant la fin des cours d'IA. L'examen d'une matière ne peut avoir lieu avant tous les cours de cette même matière.

L'événement est ensuite créé et ajouté à liste "Solution". La séance associée à celui-ci est ensuite supprimée. Cette liste d'événements constitue la solution. L'algorithme est ensuite appelé récursivement sur la liste de séances tronquée.

### 3) Prédicats utilitaires

\+ : entrée
\- : sortie
? : entrée ou sortie

### *a) Création d'un jeu de données dynamiques*

Au départ, nous créions notre base de faits à la main :

```
seance(s1).
anime(s1,hlecapitaine).
etc.
```

Cependant, cette méthode n'est pas viable pour de gros jeux de données comme l'ensemble des séances d'un semestre d'un département de l'école.

Nous avons donc choisi de créer notre base de données dynamiquement grâce au prédicat "`assertz`". Ainsi, au lancement de la consultation de notre base de faits, nous parcourons tous les blocs séances et créons dynamiquement chaque prédicat relatif à une séance (relations avec une matière, professeur, groupes etc.)

*Exemple d'un prédicat seances :*

```
seances(multimedia, cm, [id4], [jpguedon],
    [s1,s6,s29,s63,s64]).
```

Création dynamique :

```
creerSeances(Matiere,Type,Groupes,Profs,[]).
creerSeances(Matiere, Type,Groupes, Profs, [Seance|Y]):-
    assertz(seance(Seance)),
    assertz(estEnseigne(Seance,Matiere)),
    forall(member(Prof, Profs),
        (
               enseigne(Prof,Matiere),
               assertz(anime(Seance, Prof));
               assertz(anime(Seance, Prof)),
               assertz(enseigne(Prof,Matiere)) % merde
        )
    ),
    forall(member(Groupe, Groupes),
        (
               etudie(Groupe,Matiere),
               assertz(assiste(Groupe, Seance));
               assertz(assiste(Groupe, Seance)),
               assertz(etudie(Groupe, Matiere)) % merde
        )
    ),
    assertz(typeSeance(Seance,Type)),
    creerSeances(Matiere,Type,Groupes, Profs,Y).


:- forall(
       seances(Matiere,Type,Groupes,Profs,Seances),
       (
```

```
        creerSeances(Matiere,Type,Groupes,Profs,Seances)
        )
).
```

Cette méthode a plusieurs avantages :

- elle est plus pratique et offre un gain de temps non négligeable : il est bien plus rapide de créer notre base de faits de cette manière plutôt que d'écrire chaque relation liant une séance à une matière, un professeur, des groupes etc.
- la consultation de ces clauses est extrêmement rapide, puisqu'elles font alors partie du programme.
- elles nous permettent de re-planifier des séances n'ayant pas pu se tenir tout en continuant à respecter les règles générales (c'est ici le sujet de la question subsidiaire).

**Attention :** Le jeu de données étant créé dynamiquement, il ne faut pas "`consult`" deux fois notre fichier. Dans le cas échéant, des conflits dans la base de faits apparaîtraient, empêchant notre algorithme de déterminer une solution.

### *b) Manipulation de données*

Nous utilisons de nombreux prédicats permettant de manipuler les données. Ces prédicats sont généralement utilisés dans les contraintes (qui sont détaillées dans la partie suivante).

Génération de dates : `date(?Jour, ?Mois).`
Ce prédicat peut générer une date valide ou bien vérifier qu'une date l'est. Il est utilisé dans notre algorithme pour effectuer un choix non déterministe sur la date.

Génération d'une liste de séances : `makeSeances(-ListeDeSeances).`
Ce prédicat unifie la liste précisée avec la totalité des séances déclarées (dynamiquement) dans la base de faits.

Somme des effectifs d'une liste de groupe : `sommeEffectif(+ListeGroupes, -Total).`
Il effectue la somme des effectifs de chaque groupe présent dans la liste donnée en entrée. Pour cela, il utilise le prédicat `taille(+Groupe, -TailleGroupe)` qui comme son nom l'indique, permet notamment de récupérer la taille d'un groupe. Le résultat est unifié dans la variable Total.

Intersection entre deux listes : `better_intersection(+List1, +List2).`
Ce prédicat vérifie simplement si un élément se recoupe dans deux listes (c'est-à-dire qu'il est présent dans les deux listes).

Test d'incompatibilité entre deux liste de groupes : `test_incompatibilite(+List1, +List2).`
Ce prédicat peut être vu comme une extension du précédent. Il vérifie si au moins un groupe de la première liste est incompatible avec au moins un groupe de la seconde.

Il utilise le prédicat `incompatibiliteSymetrique(?Element1,?Element2)` qui vérifie simplement si un groupe est incompatible avec un autre.

Différence de plage entre deux évènements :
`difference_plage(+Plage1, +Jour1, +Mois1, +Plage2, +Jour2, +Mois2, +Resultat).`
Ce prédicat effectue la différence de temps entre deux événements. Elle est exprimée en nombre de plages et est ensuite unifié dans Résultat.

Ecriture d'un événement : `ecrireSolution([+Seance, +Salle, +Plage, +Jour, +Mois])`
Il permet un affichage plus agréable et parlant d'un événement. L'ensemble des champs doit être spécifié. Nous n'avons malheureusement pas écris correctement les jours d'une semaine (Lundi, mardi etc.) il faut ici se contenter du numéro du jour travaillé du mois (compris entre 1 et 20).

*Exemple :*

```
Date : Jour numéro 2 de Janvier sur le créneau 09:45 - 11:00
Seance : s63
Type de la séance : cm
Matière : multimedia
Groupes : [id4]
Enseignants : [jpguedon]
Salle : a1
```

`verificationEs(+Seance,+Salle,+Plage,+Jour,+Mois, [+Event|+Es])` :
Il permet de vérifier un ensemble de contraintes nécessitant un parcours des événements déjà planifiés. Pour ce faire, il utilise le prédicat `verificationE(+Seance,+Salle,+Plage,+Jour,+Mois,+Event)` (détaillé dans la suite) sur la tête de la liste et rappelle ensuite ce même prédicat sur la queue de la liste.

`verificationE(+Seance,+Salle,+Plage,+Jour,+Mois,+Event)` :
Ce prédicat est utilisé afin de vérifier des contraintes entre deux événements.

### *c) Prédicats déterministes : les contraintes*

`contrainteCM(+Seance,+Plage)` :
Elle se réalise en deux temps :

1. Si la séance n'est pas un CM, pas de problèmes
2. Dans le cas contraire on vérifie que la plage n'est pas contraignante

`contrainteJeudi(+Jour,+Plage)` :
Elle se réalise également en deux temps :

1. Si on n'est pas jeudi, pas de contraintes
2. Dans le cas contraire, on vérifie que la plage n'est pas située dans l'après midi

`contrainteUsage(+Seance,+Salle)` :

On vérifie simplement que le "type" de la séance est identique au "type" de la salle.

`contrainteTailleSalle(+Seance,+Salle)` :
Elle se réalise en trois temps :

1. Récupération de la capacité de la salle
2. Calcul du nombre de personnes qui assistent à la séance
3. On regarde sile résultat issu de la seconde étape est inférieur à la capacité de la salle

`contrainteSalleLibre(+Plage,+Jour,+Mois,+Salle,+Solution)` :
On vérifie simplement dans la liste "Solution" qu'aucun événement n'a été planifié sur cette date et dans la même salle.

*Les contraintes suivantes nécessite un parcours de la solution. On compare l'événement que l'on cherche à planifier avec tous les éléments de la liste.*

`contrainteEnseignant(+S1,+R1,+P,+J,+M, [+S2,+R2,+P,+J,+M])` :
Elle vérifie que l'enseignant (ou le groupe d'enseignants) associé à la séance n'en administre pas une à cet même instant.

`contrainteGroupe(+S1,+R1,+P,+J,+M, [+S2,+R2,+P,+J,+M])` :
Elle vérifie que deux groupes incompatibles n'ont pas cours sur le même créneau.

`contrainteOrdonnancement(+S1,+P1,+J1,+M1,[+S2,_,+P2,+J2,+M2])` :
Cette contrainte vérifie que les cours s'enchaînent dans le "bon" ordre. En effet, certains cours doivent se suivre. Par exemple, aucun examen ne doit être programmé avant que tous les cours de la matière soient terminés.

1. Si il n'y a pas de relation de séquencement, alors on passe.
2. Sinon, suivant l'ordre de séquencement des 2 séances, (la séance 1 suit la séance 2 ou inversement), on calcule la différence des plages et vérifions que le résultat se trouve bien entre la durée minimum et maximum de temps qui doit s'écouler entre les 2 séances (par défaut min = 6 créneaux, max = 18). Si la valeur est bien comprise entre ces 2 extrêmes, alors la contrainte est validée et nous pouvons continuer. Dans le cas contraire, un nouveau choix de plage horaire / jour / mois doit être effectué.

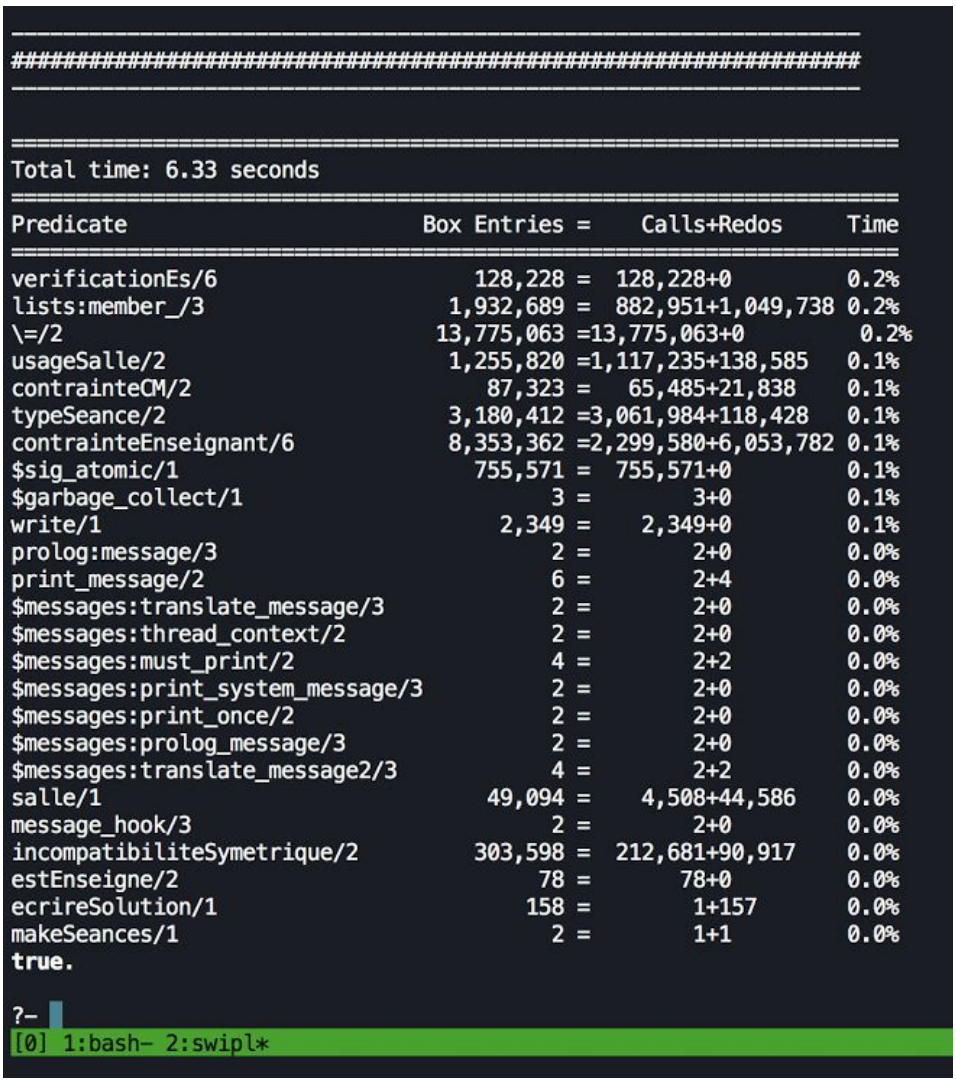
Il est important de préciser que la contrainte de séquencement de plusieurs matières et d'ordonnancement d'une matière sont rassemblées en une et unique contrainte. En effet, séquencer des matières revient à ordonnancer chaque matière. Par exemple, dire qu'une matière X précède une matière Y est équivalent à dire que la première séance de la matière Y suit la dernière de la matière X.

De plus, nous n'avons pas eu le temps d'être rigoureux sur la création du prédicat "`suitSeance(Séance1,Séance2)`". Ainsi ces prédicats ont étés écrits à la main et leur nombre est faible. En temps normal, tous les cours d'un bloc "`seances`" doivent se suivre, et nous aurions en complément lié les matières entre elles.

## 4) Test de performances

Des tests ont été effectués afin d'améliorer les performances de notre solution. Ils ont été réalisés avec la librairie "`statistics`" et consistaient à déterminer le meilleur ordonnancement des contraintes mais aussi celui des choix non déterministe. Modifier cette ordre permet notamment de diminuer (ou augmenter selon les cas) le facteur de branchement et donc de modifier la profondeur maximale de l'arbre. Cela a pour conséquence directe la modification de l'espace de recherche que l'on cherche à minimiser. Ces tests ont été réalisés à la main.

L'utilisation du prédicat "`profile`" sur notre prédicat "`faire_planification`" nous donne les statistiques suivantes :



```
________________________________________________________________
################################################################
________________________________________________________________


================================================================
Total time: 6.33 seconds
================================================================
Predicate                      Box Entries =    Calls+Redos     Time
================================================================
verificationEs/6                 128,228 =  128,228+0         0.2%
lists:member_/3                1,932,689 =  882,951+1,049,738 0.2%
\=/2                          13,775,063 =13,775,063+0          0.2%
usageSalle/2                   1,255,820 =1,117,235+138,585   0.1%
contrainteCM/2                    87,323 =   65,485+21,838    0.1%
typeSeance/2                   3,180,412 =3,061,984+118,428   0.1%
contrainteEnseignant/6         8,353,362 =2,299,580+6,053,782 0.1%
$sig_atomic/1                    755,571 =  755,571+0         0.1%
$garbage_collect/1                     3 =        3+0         0.1%
write/1                            2,349 =    2,349+0         0.1%
prolog:message/3                       2 =        2+0         0.0%
print_message/2                        6 =        2+4         0.0%
$messages:translate_message/3          2 =        2+0         0.0%
$messages:thread_context/2             2 =        2+0         0.0%
$messages:must_print/2                 4 =        2+2         0.0%
$messages:print_system_message/3       2 =        2+0         0.0%
$messages:print_once/2                 2 =        2+0         0.0%
$messages:prolog_message/3             2 =        2+0         0.0%
$messages:translate_message2/3         4 =        2+2         0.0%
salle/1                           49,094 =    4,508+44,586    0.0%
message_hook/3                         2 =        2+0         0.0%
incompatibiliteSymetrique/2      303,598 =  212,681+90,917    0.0%
estEnseigne/2                         78 =       78+0         0.0%
ecrireSolution/1                     158 =        1+157       0.0%
makeSeances/1                          2 =        1+1         0.0%
true.

?-
[0] 1:bash- 2:swipl*
```

Nous cherchons évidemment à diminuer le temps total.

## 5) Heuristiques

Ordonnancement :

Nous avons tenté de mettre en place une heuristique lié à l'ordonnancement.

Nous sommes partis du constat suivant :

Si nous ajoutons une séance à la solution et qu'elle devait suivre une autre séance non planifiée ; l'algorithme va alors mettre énormément de temps avant de revenir jusqu'au choix d'ajout de cette dite séance dans la solution (via backtracking).

L'idée est donc d'attribuer un indice de priorité à chaque séance suivant le nombre de cours qui doivent la précéder. A chaque fois qu'un cours doit être précédé d'un autre, son indice augmente de 1. Si cet autre cours doit être précédé d'un autre cours, alors l'indice prend la valeur 2 et ainsi de suite. Les cours qui ont donc le plus de cours avant eux, vont avoir un indice élevé. Une fois les indices calculés, on trie la liste des séances à planifier par ordre croissant d'indice. Les cours les plus "contraints" seront donc situés à la fin et les problèmes de mauvais ajout de séance à la solution seront donc drastiquement réduit.

Prédicat récursif qui calcule l'indice de priorité :

```
indicePriorite(S, 0) :- \+ suitSeance(S,_,_,_).
indicePriorite(S, N) :- suitSeance(S, Y,_,_), indicePriorite(Y, N1), N is N1 +1.
```

Règles pour réaliser le predsort :

```
triIndice(<, Seance1, Seance2) :-
     indicePriorite(Seance1, N1), indicePriorite(Seance2, N2), N1 < N2.

triIndice(>, Seance1, Seance2) :-
     indicePriorite(Seance1, N1), indicePriorite(Seance2, N2), N1 > N2.

triIndice(>, Seance1, Seance2) :-
     indicePriorite(Seance1, N1), indicePriorite(Seance2, N2), N1 = N2.
```

Application du predsort sur la liste de séances créées :

```
faire_planification(Solution):-
     makeSeances(ListeSeances),
     predsort(triIndice,ListeSeances,ListeTriee),
     planifier(ListeTriee,Solution).
```

## 6) Pistes d'améliorations

Journées plus légères :

A cet instant donné, notre algorithme complète toutes les plages disponibles une à une tant qu'il n'y a pas de conflit. Des journées répétées de 9 heures ne sont pas envisageable en réalité afin de préserver la santé des étudiants.

Jeu de données :

Bien que le jeu de données se génère en partie automatiquement, de longues phases d'écriture sont encore nécessaires.

Ajout de spécificités :

Notre programme n'est pas capable d'ajouter les évènements occasionnels et plus spécifiques comme les tiers temps, les indisponibilité etc. De plus, les créneaux d'anglais et de sport doivent toujours être planifié sur le même jour et plage de la semaine, ce qui n'est actuellement pas le cas.

Ajout d'heuristiques :

Nous souhaitions ajouter plus d'heuristiques afin de réduire au maximum l'espace de recherche et obtenir une "bonne solution" dans les plus brefs délais. Cependant, nous avons passé trop de temps sur l'ordonnancement, nous empêchant de nous concentrer sur d'autres heuristiques.

## 7) Question subsidiaire

Comme nous disposons d'une base de données dynamique, il nous est facile d'ajouter des séances avec le prédicat "`assert`".

## 8) Difficultés rencontrées

Même si nous avions déjà abordé le Prolog, nous ne comprenions pas réellement comment se fait l'unification des variables et les callbacks. Cette incompréhension nous a fait perdre un temps précieux dans l'écriture des contraintes.

Nous avons également effectué de nombreuses modifications dans notre jeu de données : changement de la mesure du temps, base de données dynamique etc. Chaque bouleversement a entraîné une réécriture partielle, plus ou moins conséquente, de l'algorithme de solution.

Dernièrement, nous avons mis un temps conséquent à trouver une potentielle heuristique (et d'autant plus à l'implémenter).