République Algérienne Démocratique et Populaire

Ministère de l'enseignement supérieur et de recherche scientifique

Université Larbi Ben M'hidi OEB

FACULTE DES SCIENCES EXACTES ET SCIENCES DE LA NATURE ET DE LA VIE

Département de Mathématiques et Informatique

Mémoire présenté pour l'obtention du diplôme de

Master en Informatique

Option : Architecture Distribuée

Titre du Mémoire :

Réalisation d'une application pour la génération automatique d'emploi du temps universitaire

Présenté par :

Kahil Moustafa Sadek

Devant le jury composé de :

Président	Dr BOUNEB Zineddine	Université d'OEB
Examinateur	Pr MOUKHATI Farid	Université d'OEB
Encadreur	Dr MELLAL Nassima	Université d'OEB

Année universitaire : 2015– 2016

Remerciements

Ce travail n'aurait jamais été mené à terme sans le soutien de DIEU *le tout puissant qui m'a donné la force et la volonté pour le réaliser.*

J'exprime mes remerciements les plus sincères à Mme N. MELLAL d'avoir assuré la direction de ce travail, de m'avoir encadré et de m'avoir guidé.

Je tiens à remercier les membres du jury Mr Z. BOUNEB et Mr F. MOUKHATI d'avoir accepté d'examiner ce mémoire.

J'adresse ma gratitude à tous ceux qui ont contribué, de près ou de loin à réaliser ce mémoire par leur présence et leurs conseils.

Dédicaces

Tous les mots que je connais ne sauraient exprimer ma gratitude pour mon père Abdelaziz qui a été à mes côtés tout au long de la préparation de ce mémoire, à ma maman dont les prières intenses m'ont accompagné. Puissent-ils être contents de mon œuvre.

Que ce modeste travail soit l'expression des vœux que vous n'avez cessé de formuler dans vos prières. Que Dieu vous préserve et vous procure santé et longue vie.

Une pensée particulière pour mes frères et sœurs dont les railleries m'ont malgré tout encouragé. Et surtout Meriem.

Je ne saurais oublier mes neveux et nièces dont la présence et les caprices ont agrémenté les jours de travail. Je citerai notamment Lina et Ahmed Yacine.

J'exprime mes dédicaces à chers mes amis, à tous mes proches ainsi qu'à mes enseignants pour leur dévouement et leur bienveillance.

Votre enfant, parent, ami, élève

Moustafa Sadek

Résumé

Le problème d'emploi du temps peut être vu comme une instance des problèmes d'ordonnancement des tâches. Ce type consiste à distribuer des ressources sur des tâches selon un ordre adéquat et donc dans un temps approprié, et ce tout en satisfaisant un ensemble de contraintes concernant les différentes parties constituant le problème. D'où nous apparaît la notion de problèmes de satisfaction des contraintes.

La construction d'emploi du temps universitaire consiste à organiser des rencontres (des séances de cours, de TD et de TP) entre les enseignants et les étudiants dans des salles de différentes catégories (Amphithéâtres, salles de TD et salle de TP) et durant des périodes hebdomadairement fixes.

Dans ce travail, nous nous intéressons à la résolution automatique du problème d'emploi du temps universitaire. Pour ce fait, nous proposons une approche basée sur la résolution des CSP (problèmes de satisfaction des contraintes).

Mots clés : Ordonnancement, Emploi du temps, Problèmes de satisfaction des contraintes, Programmation logique avec contraintes.

Abstract

Timetable problems is an instance of task scheduling problems which consist to allocate resources to tasks in an appropriate order and therefore appropriate time, with satisfying constraints of different problem's sections. Hence the concept of constraint satisfaction problems.

The university timetabling consists to organize meetings (lectures) between instructors and students in different rooms types (amphitheaters, tutorial classes rooms and practical works rooms) and during weekly fixed timeslots.

In this work, we aim to automating the university timetabling. We propose a CSP based approach to realize this system.

Keywords: Scheduling, Timetabling, Constraint satisfaction problems, Constraint logic programming.

ملخص

يعتبر مشكل الجدولة الزمنية من المشاكل المعقدة في ميدان الإعلام الآلي. هذا النوع يندرج ضمن مشاكل جدولة المهام التي تعمل على إيجاد لإدارة استعمال وتوزيع الموارد المشتركة على المهام بالشكل الأمثل وفق القيود الموضوعة. ومن هنا جاء مفهوم البرمجة المقيدة.

مشكلة الجدولة الزمنية بالنسبة للجامعات هي حالة معقدة من المشاكل السابقة، حيث تنص على برمجة حصص (المحاضرات، الأعمال التطبيقية والأعمال الموجهة) بين الأساتذة والطلاب في قاعات بمختلف أنواعها: المدرجات الخاصة بالمحاضرات، القاعات الخاصة بالأعمال التطبيقية والقاعات الخاصة بالأعمال الموجهة خلال مدة زمنية محددة، وهذا مع مراعاة عدة قيود تخص الأساتذة والطلاب والقاعات

سنحاول من خلال هذا العمل اقتراح طريقة للجدولة الزمنية آليا. وسنعتمد من أجل ذلك البرمجة المنطقية المقيدة.

الكلمات المفتاحية: الجدولة، التوقيت، البرمجة المنطقية المقيدة.

Sommaire

Liste de figures

Liste des algorithmes

Notations

EDT	Emploi Du Temps
CSP	Problèmes de Satisfaction de Contraintes
DCSP	Problèmes de Satisfaction Distribuée des Contraintes
PL	Programmation Logique
PPC	Programmation Par Contraintes
CLP	Programmation Logique par Contraintes
RO	Recherche Opérationnelle
IA	Intelligence Artificielle
SA	Recuit Simulé
LS	Recherche Locale
GA	Algorithme Génétique
SMA	Systèmes Multi-Agents
SGBD	Système de Gestion de Bases de Données

Introduction

Pour toute institution soucieuse de rentabilité et d'organisation perfectionnée, élaborer un emploi du temps est un élément essentiel et nécessaire. Que ce soit pour l'entreprise ou l'école, il s'agit toujours de maîtriser le fonctionnement de l'institution à travers le bon emploi des ressources humaines notamment. Et cela est une tâche autant ardue qu'indispensable qui requiert l'emploi de l'outil informatique. La réalisation d'un tel système est donc devenue un choix incontournable pour l'automatisation des emplois du temps dans le but de faire gagner le plus d'heures de travail pour toutes les parties d'une activité donnée. En outre, l'objectif est plus ambitieux car il tend à fournir des solutions presque optimales et rapides à toute sorte de contraintes en vue tout aussi bien d'accroître la production que la qualité des services.

La difficulté de construire des emplois du temps réside dans la diversification des normes et des problèmes posés par les utilisateurs ainsi que dans les caractéristiques propres à chaque institution concernée. Dans le secteur éducatif et précisément celui universitaire, l'accroissement de nombre d'étudiants et d'enseignants ainsi que le nombre restreint des structures d'accueil imposent des contraintes que l'on doit gérer et agencer. Ceci rend très difficile de trouver une solution générale aux problèmes d'emploi du temps universitaire, ce qui implique la nécessité de plus de recherche dans ce domaine, car trouver un horaire respectant toutes les contraintes n'est aisé même avec les approches proposées pour la résolution informatique de ce problème.

Nous allons essayer, à travers ce mémoire, de proposer une approche de résolution du problème d'emploi du temps universitaire.

Le premier chapitre de ce mémoire commence par étudier superficiellement le problème de l'ordonnancement ainsi quelques instances auquel elles appartiennent, présente ensuite le problème d'emploi du temps universitaire et les différentes approches existantes dans la littérature liées à sa résolution et cite enfin quelques travaux proposés dans la communauté scientifiques. Le deuxième chapitre est un petit manuel contenant des généralités sur le paradigme de programmation logique avec contraintes ainsi qu'une initiation au langage que nous utilisons au cours de notre approche de résolution. Le troisième chapitre traite de la phase de conception de notre système après le positionnement des différents besoins qu'il doit fournir. Le dernier chapitre aborde la réalisation de l'application avec une brève description et une petite expérimentation et termine par une comparaison avec une autre application spécialisée dans la résolution du même problème. En conclusion, nous citons quelques perspectives contenant des voies d'amélioration de notre approche.

CHAPITRE 01 : ETAT DE L'ART

1. Introduction

Créer un calendrier, dans n'importe quel domaine de production, est devenu une étape essentielle, elle doit passer par un ensemble de phases bien temporairement ordonnées et structurées. Prenons, par exemple, le cas de fabrication des meubles. Si on veut fabriquer une table, il faut, après avoir pris les mesures nécessaires et apporté du bois, construire d'abord le haut et le dessous de la table et créer les pieds, fixer ensuite ces derniers à l'envers de la table chacun dans son coin, vérifier la stabilité de la table et la peindre enfin. Quand ce travail est fait par un groupe, chaque individu ou sous-groupe se charge d'en effectuer une tâche spécifique. Il est devenu par conséquent important de dépendre d'une sorte de synchronisation pour que toutes ces étapes soient ordonnées.

Afin de rendre facile l'organisation de ces phases, il est obligatoirement préféré de suivre un processus qui se charge de ce « job ». Un terme fortement relatif à ce processus dans tous les domaines (mécanique, statistiques, informatique, …) est l'ordonnancement.

Le problème d'emploi du temps représente un cas concret de l'ordonnancement où entrent en jeux plusieurs tierces effectuant des tâches et utilisant des ressources limitées (les locaux) dans un temps limité tout en satisfaisant un ensemble de contraintes bien précises.

Dans ce chapitre, nous allons présenter quelques concepts sur l'ordonnancement. Ensuite, nous présenterons le problème d'emploi du temps et plus précisément le problème d'emploi du temps universitaire. Et nous finirons par citer quelques travaux liés à la résolution de ce problème.

2. L'ordonnancement

Dans les systèmes organisés, notamment ceux qui comportent un nombre important de ressources, la planification des tâches est un mécanisme lourd, difficile et complexe ; le processus de génération doit couvrir le traitement de plusieurs phases afin de satisfaire de différentes contraintes liées à ces ressources. C'est pour cela que nous devons avoir recours à l'ordonnancement.

2.1- Définitions de l'ordonnancement

Plusieurs définitions de l'ordonnancement ont été proposée dans la communauté scientifique. Mais on remarque que toutes ces définitions s'accordent les mêmes éléments qui le composent. Nous présentons ici deux définitions la première définie l'ordonnancement de façon superficielle et la seconde raffine la signification de ce concept au domaine informatique.

Définition 1 (J.Carlier, P.Chrétienne, J.Erschler, C.Hanen, P.Lopez, E.Pinson, M-C.Portmann, C.Prins, 1988)

Ordonnancer un ensemble de tâches, c'est programmer leur exécution en leur allouant les ressources requises et en fixant leurs dates de début. La théorie de l'ordonnancement traite de modèles mathématiques mais analyse également des situations réelles fort complexes ; aussi le développement de méthodes utiles ne peut-il être que le fruit de contacts entre la théorie et la pratique.

Les problèmes d'ordonnancement apparaissent dans tous les domaines de l'économie : l'informatique (les tâches sont les programmes ; les ressources sont les processeurs, la mémoire,

...), la construction (suivi de projet), l'industrie (activités des ateliers en gestion de production et problèmes de logistique), l'administration (emplois du temps).

Définition 2 (Wikipedia, Théorie de l'ordonnancement, 2016)

Un problème d'ordonnancement consiste à organiser dans le temps la réalisation de tâches, compte tenu de contraintes temporelles (délais, contraintes d'enchaînement) et de contraintes portant sur la disponibilité des ressources requises.

En production (manufacturière, de biens, de service), on peut le présenter comme un problème où il faut réaliser le déclenchement et le contrôle de l'avancement d'un ensemble de commandes à travers les différents centres composant le système.

Un ordonnancement est défini par le planning d'exécution des tâches (« ordre » et « calendrier ») et d'allocation des ressources et vise à satisfaire un ou plusieurs objectifs. Un ordonnancement est très souvent représenté par un diagramme de Gantt[1].

D'après ces deux définitions, nous pouvons estimer que l'ordonnancement sert à planifier des tâches dans un temps et un ordre précis selon la disponibilité de ressources requises à ce fait tout en respectant les contraintes liées aux tâches et ressources. Nous allons maintenant projeter ces notions en un peu plus de détail.

2.2- Notions : (les ressources et les tâches) (J.Carlier, P.Chrétienne, J.Erschler, C.Hanen, P.Lopez, E.Pinson, M-C.Portmann, C.Prins, 1988)

Dans un problème d'ordonnancement s'introduisent deux notions fondamentales : les ressources et les tâches. Une ressource est un moyen, technique ou humain, dont la disponibilité limitée ou non est connue a priori. Une tâche est un travail élémentaire dont la réalisation nécessite un certain nombre d'unités de temps (sa durée) et d'unités de chaque ressource.

La résolution d'un problème d'ordonnancement doit concilier deux objectifs :

- ✓ L'aspect statique consiste à générer un plan de réalisation des travaux sur la base de données prévisionnelles.
- ✓ L'aspect dynamique consiste à prendre des décisions en temps réel compte tenu de l'état des ressources et de l'avancement dans le temps des différentes tâches.

Les données d'un problème d'ordonnancement sont les tâches et leurs caractéristiques, les contraintes potentielles, les ressources et la fonction « économique » (la fonction objectif)[2]. On note en général $\mathbf{I} = \{\mathbf{1}, \mathbf{2}, \dots, \mathbf{n}\}$ l'ensemble des tâches et $\mathbf{p_i}$ la durée de la tâche **i** si cette durée ne dépend pas des ressources qui lui sont allouées. Souvent, une tâche **i** ne peut commencer son exécution avant une date de disponibilité notée $\mathbf{r_i}$ et doit être achevée avant une date échue $\mathbf{d_i}$. Les dates réelles de début et de fin d'exécution de la tâche **i** sont notées respectivement $\mathbf{t_i}$ et $\mathbf{C_i}$. Les tâches sont souvent liées entre elles par des relations d'antériorité (de précédence). Si ce n'est pas le cas on dit qu'elles sont indépendantes.

La contrainte d'antériorité la plus générale entre deux tâches **i** et **j**, appelée contrainte potentielle, s'écrit sous la forme $\mathbf{tj - ti \geq aij}$; elle permet d'exprimer la succession simple ($\mathbf{aij = pi}$) et de nombreuses variantes.

[1] Il s'agit d'une représentation d'un graphe connexe, valué et orienté, qui permet de représenter graphiquement l'avancement du projet.

[2] Elle associe une valeur à une instance d'un problème d'optimisation. Le but du problème d'optimisation est de minimiser ou de maximiser cette fonction jusqu'à l'optimum.

2.3- Classes des problèmes d'ordonnancement

Il existe plusieurs catégories des problèmes d'ordonnancement. Ils se diffèrent selon des critères liés aux notions que nous venons de définir (tâches et ressources). Nous en citons ici brièvement quelques types très fameux en Informatique.

2.3-1. Problèmes centraux *(J.Carlier, P.Chrétienne, J.Erschler, C.Hanen, P.Lopez, E.Pinson, M-C.Portmann, C.Prins, 1988)*

Dans le problème central de l'ordonnancement, il s'agit d'ordonnancer, en une durée minimale des tâches soumises à des contraintes de succession et de localisation temporelle. Ces contraintes sont résumées par un graphe **potentiels-tâches** $\mathbf{G} = (\mathbf{X}, \mathbf{U})$ où **X** est l'ensemble des tâches complété par une tâche fictive début notée **0** et une tâche fictive fin notée $*$, et **U** est formé des arcs associés aux contraintes potentielles, l'arc **(i,j)** évalué par $\mathbf{a_{ij}}$ correspondant à la contrainte potentielle $\mathbf{t_j - t_i \geq a_{ij}}$. Un ordonnancement est un ensemble $\mathbf{T} = \{\mathbf{t_i} / \mathbf{i} \in \mathbf{X}\}$ de dates de début des tâches telles que la date de début de la tâche fictive **0** soit zéro et que, pour tout arc **(i,j)**, $\mathbf{t_j - t_i \geq a_{ij}}$. Un tel ordonnancement existe si et seulement si **G** ne contient pas de circuit de valeur strictement positive.

2.3-2. Problèmes à ressource unique *(J.Carlier, P.Chrétienne, J.Erschler, C.Hanen, P.Lopez, E.Pinson, M-C.Portmann, C.Prins, 1988)*

Le principe de base de ce type de problèmes est de déterminer sur la ressource un séquencement optimal de n tâches disponibles à l'instant **0** et de durées $\mathbf{p_1}$, **…,** $\mathbf{p_n}$. Des algorithmes polynomiaux se réduisant à des tris permettent de résoudre le problème de base pour les critères de minimisation des encours (l'encours de la tâche **i** étant $\mathbf{C_i}$ ou $\mathbf{w_i C_i}$, si on attribue un poids $\mathbf{w_i}$ à la tâche **i**) et minimisation du plus grand retard. Dans le premier cas, on doit ordonner les tâches dans le sens des $\mathbf{p_i/w_i}$ croissants (règle de SMITH) ; dans le second, dans le sens des $\mathbf{d_i}$ croissants (règle de JACKSON). L'algorithme polynomial de HODGSON permet de minimiser le nombre de tâches en retard, il consiste à placer les tâches, une par une, dans l'ordre des dates échues croissantes, en rejetant la tâche déjà placée de plus grande durée à la fin de l'ordonnancement si un retard apparaît. Le problème est NP-difficile pour la somme pondérée des retards, y compris lorsque tous les poids sont égaux. On trouve dans la littérature des travaux présentant des méthodes exactes (arborescentes ou de programmation dynamique) ou des méthodes approchées résolvant différents problèmes (à une ressource) dérivés du problème de base en ajoutant des contraintes.

2.3-3. Problèmes à ressources multiples *(J.Carlier, P.Chrétienne, J.Erschler, C.Hanen, P.Lopez, E.Pinson, M-C.Portmann, C.Prins, 1988)*

Les problèmes d'ordonnancement à contraintes de ressources renouvelables (multiples) sont souvent appelés problèmes d'ateliers car on les rencontre dès qu'il faut gérer la production. Toutefois il s'agit de problèmes très généraux car ils apparaissent dans d'autres contextes, en particulier en Informatique.

Dans le cas des problèmes d'atelier, une tâche est une opération, une ressource est une machine et chaque opération nécessite pour sa réalisation une machine. Dans le modèle de base de l'ordonnancement d'atelier, l'atelier est constitué de **m** machines, **n** travaux (jobs), disponibles à la date **0**, doivent être réalisés, un travail **i** est constitué de $\mathbf{n_i}$ opérations, l'opération j du travail

i est notée **(i,j)** avec $(\mathbf{i},\mathbf{1}) < (\mathbf{i},\mathbf{2}) < \ldots < (\mathbf{i},\mathbf{ni})$ si le travail **i** possède une gamme (**A < B** signifie **A précède B**). Une opération $(\boldsymbol{i},\boldsymbol{j})$ utilise la machine $\boldsymbol{mi},\boldsymbol{j}$ pendant toute sa durée $\boldsymbol{pi},\boldsymbol{j}$ et ne peut être interrompue.

Une classification peut s'opérer selon l'ordre d'utilisation des machines pour fabriquer un produit **(*gamme de fabrication*)**. On rencontre : des ateliers à *cheminement unique* où toutes les gammes sont identiques **(*flow-shop*)**, des ateliers à *cheminements multiples* où chaque produit ou famille de produits possède **(*job-shop*)** ou ne possède pas **(*open-shop*)** une gamme spécifique. Les dates de début des opérations **(i,j)** constituent les inconnues du problème et leur détermination définit l'ordonnancement. Les contraintes sont de type disjonctif ; le choix d'une séquence sur une machine résout donc les conflits d'utilisation de la machine.

Le modèle de base peut être étendu de diverses façons en prenant en compte, par exemple l'existence de dates échues impératives ($\mathbf{d_i}$), l'interruption possible d'une opération (avec ou sans mémoire du travail effectué), les temps de préparation, une disponibilité des machines variable dans le temps, des ressources à capacité non unitaire, l'utilisation de plusieurs ressources pour une opération, ...

2.3-4. Problèmes cycliques

Ces problèmes sont présents dans plusieurs domaines car leur résolution consiste à organiser des opérations au fil du temps. Ces opérations peuvent être des tâches informatiques, des étapes d'un projet de construction, des opérations dans un processus de production, … etc. Les contraintes dans un problème d'ordonnancement, en plus de l'ordre de priorité entre les opérations, sont dues à l'attribution de ressources limitées. Ces ressources peuvent être des processeurs, des employés, des machines. L'aspect cyclique de ces problèmes est relatif au fait que les opérations sont exécutées plusieurs fois.

Dans un problème d'ordonnancement cyclique, un ensemble de tâches dites génériques est exécuté une infinité de fois, l'objectif étant généralement de minimiser le temps entre deux occurrences d'une même tâche, une occurrence d'une tâche étant une réalisation de celle-ci.

Les problèmes d'ordonnancement cycliques se regroupent, eux-aussi, en plusieurs catégories. Nous en jetons brièvement un coup d'œil sur quelques-unes.

2.3-4.1. Problèmes cycliques de base (Ben Rahou, 2013)

Le problème d'ordonnancement cyclique de base (General Basic Cyclic Scheduling Problem) (GBCSP) est caractérisé par un ensemble de tâches élémentaires (ou génériques) $T = \{\mathbf{1},\ldots,\mathbf{n}\}$ qui seront répétées une infinité de fois. Chaque tâche $\boldsymbol{i}$ a une durée $\boldsymbol{pi}$ on note $< \mathbf{i},\mathbf{k} >$ la $\mathbf{k}^{\text{ème}}$ occurrence de **i** telle que $\mathbf{k} \in N$. Dans un GBCSP, les tâches sont liées par des contraintes de précédence dites uniformes. Résoudre un GBCSP revient à trouver un ordonnancement cyclique **s** qui détermine pour chaque occurrence $< \mathbf{i},\mathbf{k} >$, sa date de début $\mathbf{t}(\mathbf{i},\mathbf{k})$ tout en minimisant le temps de cycle asymptotique $\boldsymbol{\alpha}$. (Ben Rahou, 2013)

L'objectif est la minimisation du temps de cycle asymptotique **a**. Les contraintes représentent l'ensemble des contraintes de précédence. Les contraintes imposent le non dépassement entre les différentes occurrences d'une même tâche, ces contraintes de non dépassement sont incluses dans les contraintes de précédences si on considère que $\mathbf{Hii} = 1$ pour toute tâche i (Hij est la hauteur des occurrences entre les tâches i et j).

2.3-4.2. Ordonnancement k-cyclique (Ben Rahou, 2013)

Dans un ordonnancement 1-cyclique, la différence entre les dates de début de deux occurrences différentes d'une même tâche est égale au temps de cycle α, ainsi, dans un intervalle de temps de longueur α chaque tâche est traitée exactement une fois. En revanche, dans un ordonnancement *K*-cyclique la différence entre la date de début de l'occurrence $< \boldsymbol{i, n} >$ et la date de début de l'occurrence $< i, (n + K) >$ est égale au temps de cycle $\boldsymbol{\alpha_K}$. Par conséquent, dans un intervalle de temps de longueur α_K chaque tâche est traitée exactement *K* fois. Autrement dit, dans un ordonnancement *K*-cyclique, l'ordonnancement de *K o*ccurrences successives de chaque travail est fixe et se répète à un intervalle régulier de longueur α_K.

2.3-4.3. Ordonnancement cyclique à machines parallèles (PMCS) (Ben Rahou, 2013)

Dans un problème de machines parallèles un ensemble des opérations (tâches) $\{Oi\}, 1 \leq i \leq n$ est exécuté sur m processeurs (machines) identiques. Pour chaque exécution d'une opération *i*, il est nécessaire d'utiliser un des *m* processeurs pendant *pi* unités de temps. La préemption n'est pas autorisée. Les contraintes de ressources sont liées au nombre limité de processeurs et au nombre d'opérations qui sont exécutées à une période donnée. Un ensemble de contraintes de précédence est également à considérer. Un ordonnancement réalisable est un choix de dates de début $\{ti\}, 1 \leq i \leq n$, tel que les contraintes de ressources et de précédence soient respectées.

2.3-4.4. Problème Job-Shop cyclique (CJSP) (Ben Rahou, 2013)

Dans un CJSP, le nombre de machines est inférieur au nombre de tâches génériques. Ces machines ne sont pas identiques et il n'y a pas de problème d'affectation : toute tâche est pré-affectée à une des machines, qu'elle doit donc partager avec toutes les autres tâches également affectées à cette machine. Par conséquent, les machines jouent un rôle important et les contraintes qu'elles génèrent sont ajoutées au CJSP. L'objectif de la résolution d'un CJSP est de trouver un ordonnancement périodique ayant un temps de cycle asymptotique minimal et vérifiant toutes les contraintes de précédence et de ressource.

Un CJSP est défini comme un GBCSP auquel on ajoute des contraintes de ressources (Ben Rahou, 2013). Soit $M = \{1, \ldots, m\}$ un ensemble de machines où $m < n$. La tâche *i* est exécutée sur la machine $M(i) \in M$ sans interruption pendant toute sa durée p_i. Le chevauchement de deux occurrences de tâches différentes utilisant la même machine représente un conflit de ressources. Ce conflit est évité en ajoutant des contraintes disjonctives.

2.4- Problèmes d'ordonnancement cyclique sous contraintes de ressources (Ben Rahou, 2013)

Au-delà des problèmes d'atelier, où les ressources sont des machines ou des robots de type disjonctif, c'est à dire qu'une ressource ne peut exécuter qu'une seule tâche à un instant donné, des travaux récents considèrent des problèmes impliquant des ressources plus complexes, appelées ressources cumulatives ou discrètes selon les auteurs. Ces problèmes apparaissent par exemple dans le contexte de l'ordonnancement d'instruction au sein des compilateurs pour les architectures parallèles. Une ressource cumulative est définie par une capacité maximale instantanée (par exemple une taille mémoire) et chaque tâche demande éventuellement tout au long de son exécution un nombre limité mais possiblement supérieur à 1 d'unités de cette ressource. Ainsi à tout moment le nombre de tâches utilisant cette ressource exécutée en parallèle

est limité par le fait que la somme des demandes des tâches ne doit pas excéder la capacité de la ressource. En ordonnancement acyclique, un problème très classique impliquant ce type de ressource est le **RCPSP (Resource-Constrained Project Scheduling Problem).** Il faut noter que lorsqu'une seule ressource est disponible et que les tâches demandent chacune une seule unité, on obtient le problème à machines parallèles. Aussi, les méthodes d'ordonnancement cyclique dans ce contexte sont souvent adaptées des méthodes utilisées pour résoudre le **RCPSP** d'un côté et des techniques de résolution du **PSIP (Problème d'ordonnancement cyclique à machines parallèles)** d'un autre côté.

3. Problèmes d'emploi du temps universitaire

Nous allons d'abord présenter le problème d'emploi du temps en général puis nous le particularisons au cas de celui universitaire.

3.1- Problèmes d'emploi du temps

Le problème d'emploi du temps est l'un des problèmes de calcul difficiles dans l'ordonnancement. Dans le processus de génération d'un emploi du temps, le but est de trouver des créneaux horaires adaptés à un nombre de tâches qui nécessitent des ressources limitées. Selon la nature du problème d'emploi du temps, les contraintes peuvent se varier et donc il se pourrait qu'il y ait de nombreux objectifs différents. Dans certains cas (Ömer S. Benli, A. Reha Botsali, 2004), l'objectif est de trouver une solution réalisable dans une durée totale fixée au préalable. Dans d'autres cas, le but est de réduire la période durant laquelle les tâches doivent s'exécuter. On peut même tomber dans des cas où l'objectif est de trouver une solution qui viole un nombre minimum de contraintes.

Le problème de génération d'emploi du temps peut survenir dans de nombreux contextes différents. Il peut figurer dans le cadre de gestion des établissements de santé où il faut créer un calendrier (hebdomadaire, mensuel ou annuel) pour les employés (médecins, infirmiers, laborantins, …) qui gère les jours et heures de travail pour chacun, les jours de permanences, … etc. Dans le secteur du sport apparaît bien ce problème. L'exemple du football illustre aussi bien l'utilité de la planification d'emploi du temps : un championnat concerne un ensemble d'équipes qui font des compétitions entre elles dans des stades de football précis durant des périodes précises en respectant plusieurs contraintes. On peut en citer les contraintes de disponibilité des stades et des équipes concernées par les matchs, prendre en considération que ces équipes peuvent participer en plusieurs championnats. Enfin, ce problème se réfère évidemment à l'emploi du temps dans le domaine scolaire. L'emploi du temps universitaire, qui est notre cas d'étude, fait partie de ce dernier domaine.

3.2- Problème d'emploi du temps universitaire

Certains considèrent le problème de génération d'emploi du temps universitaire comme une instance des problèmes d'ordonnancement cyclique. Sa résolution consiste à ordonnancer les tâches d'un ensemble d'enseignants et de groupes appartenant à des sections en leur allouant un ensemble de locaux et en leur fixant des créneaux horaires, ces tâches ont un caractère cyclique. D'autres le voient comme un problème NP-complet. Ce type de problèmes est réputé complexe et sa résolution optimale exige un temps exponentiel. C'est pour cela qu'on envisage d'utiliser les heuristiques.

Ces deux visions ne sont pas contradictoires vue que la première consiste à chercher une solution réalisable à ce problème (problème d'emploi du temps vu comme un problème de recherche) et la seconde consiste à tenter de spécifier aux séances des périodes ardemment précises (problème d'emploi du temps vu comme un problème d'optimisation).

Le processus de génération d'emploi du temps universitaire pourrait être formellement défini avec une matrice binaire $Xegsjpl$ qui prend la valeur 1 si l'enseignant $\boldsymbol{e}$ peut enseigner la séance $\boldsymbol{s}$ avec le groupe $\boldsymbol{g}$ dans la période $\boldsymbol{p}$ du jour $\boldsymbol{j}$ dans le local $\boldsymbol{l}$.

Il existe essentiellement deux classes de contraintes : les contraintes dures (hard) et les contraintes souples (soft).

Contraintes dures (hard) : cette classe représente le type de contraintes dont on n'a pas le droit de les enfreindre.

Contraintes souples (soft) : c'est l'ensemble de contraintes de préférence. En les relâchant, la sortie reste toujours valide mais pas optimale.

4. Méthodes de résolution

L'objectif principal des méthodes proposées est de générer un emploi du temps valide qui, à la fois, satisfait toutes les contraintes dures et le maximum des contraintes souples qui concernent les enseignants et les étudiants.

Malgré le progrès réalisé dans les technologies informatiques (matérielles et logicielles), la question de trouver une solution formelle au problèmes d'élaboration des emplois du temps efficaces et souhaitables reste toujours posée dans la communauté scientifique.

Plusieurs travaux de recherche relatifs à ce domaine ont été réalisés. Ils reposent sur différentes stratégies. En sachant qu'il existe plusieurs critères de classification pour ces dernières (comme, par exemple, résolution mathématique / programmation par contraintes (Ömer S. Benli, A. Reha Botsali, 2004), recherche opérationnelle / intelligence artificielle (MILI, 2010)), nous préférons de les classifier sous deux catégories plus générales comportant, différemment aux autres classifications, les méthodologies les plus récentes : l'approche centralisée et l'approches distribuée.

Nous allons citer quelques travaux réalisés dans chacune des deux classes.

4.1- Approches centralisées

Toutes les approches utilisées pour la résolution du problème d'EDT de façon centralisée suivent deux : la recherche opérationnelle **(RO)** et l'intelligence artificielle **(IA)**.

4.1-1. Approches basées sur la recherche opérationnelle (RO)

En général, la recherche opérationnelle représente l'ensemble des techniques et méthodes destinées à la recherche des meilleurs résultats aux problèmes dont la résolution dépend d'une décision sur un ensemble de solutions possibles.

Plusieurs approches entrant dans le cadre de la recherche opérationnelle ont été proposées pour résoudre le problème d'EDT. Elles sont classées sous deux méthodologies très connues dans la RO : les méthodes de voisinage et les méthodes de construction.

La figure () montre les approches de résolution appartenant aux deux classes.

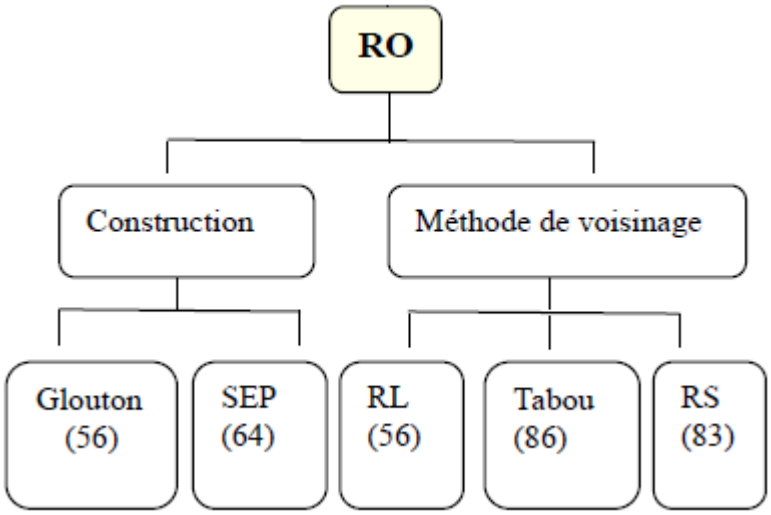

fig 1 Liste des approaches de resolution appurtenant à la recherche opérationnelle (Houria, 2012)

Dans la première classe, on trouve les méthodes gloutonnes et les SEP (Séparation et Evaluation Progressive). Tandis que la seconde classe, elle comprend trois approches : la recherche locale (RL), la recherche tabou et le recuit simulé (RS).

Nous avons choisi de présenter deux travaux basés sur les méthodes de voisinage. Le premier est basé sur la recherche locale et le second est basée sur le recuit simulé.

a. Recherche locale

Avant de décrire le travail proposé, nous allons d'abord essayer d'avoir quelques notions sur cette approche.

Définition *(Wikipedia, Recherche locale (optimisation), 2016)*

En algorithmique, la recherche locale est une méthode générale utilisée pour résoudre des problèmes d'optimisation, c'est-à-dire des problèmes où l'on cherche la meilleure solution dans un ensemble de solutions candidates. La recherche locale consiste à passer d'une solution à une autre dans l'espace des solutions candidates (*l'espace de recherche*) jusqu'à ce qu'une solution considérée comme optimale soit trouvée ou que le temps imparti soit dépassé.

Notions *(Wikipedia, Recherche locale (optimisation), 2016)*

Un algorithme de recherche locale part d'une solution candidate et la déplace de façon itérative vers une solution voisine. Cette méthode est applicable seulement si une notion de voisinage est définie sur l'espace de recherche. Par exemple, le voisinage d'un arbre recouvrant est un autre arbre recouvrant qui diffère seulement par un nœud. Pour le problème SAT[3], les voisins d'une bonne affectation sont habituellement les affectations qui diffèrent seulement par la valeur d'une variable. Le même problème peut avoir plusieurs définitions différentes de

[3] Le **problème SAT** (problème de satisfaisabilité booléenne) est le problème de décision, qui détermine s'il existe une assignation des variables propositionnelles d'une une formule de logique propositionnelle qui rend la formule vraie.

voisinage ; l'optimisation locale avec des voisinages qui limitent les changements à *k* composantes de la solution est souvent appelée *k*-optimale.

Habituellement, chaque solution candidate a plus d'une solution voisine ; le choix de celle vers laquelle se déplacer est pris en utilisant seulement l'information sur les solutions voisines de la solution courante, d'où le terme de recherche *locale*. Quand le choix de la solution voisine est fait uniquement en prenant celle qui maximise le critère, la métaheuristique prend le nom de *hill climbing* (escalade de colline).

Le critère d'arrêt de la recherche locale peut être une limite en durée. Une autre possibilité est de s'arrêter quand la meilleure solution trouvée par l'algorithme n'a pas été améliorée depuis un nombre donné d'itérations. Les algorithmes de recherche locale sont des algorithmes sous-optimaux puisque la recherche peut s'arrêter alors que la meilleure solution trouvée par l'algorithme n'est pas la meilleure de l'espace de recherche. Cette situation peut se produire même si l'arrêt est provoqué par l'impossibilité d'améliorer la solution courante car la solution optimale peut se trouver loin du voisinage des solutions parcourues par l'algorithme.

Recherche locale et emploi du temps (Résolution)

Ce travail a été présenté par (Olivia Rossi-Doria, Christian Blum,Joshua Knowles, Michael Sampels,).

Le problème consiste en :

- un ensemble de cours C à programmer en 36 intervalles de temps (6 jours de 6 créneaux horaires chacun) ;
- un ensemble de salles S dans lesquelles les événements peuvent avoir lieu ;
- un ensemble de groupes d'étudiants G qui assistent aux cours ;
- et un ensemble de caractéristiques F satisfaites par salle et exigées par les cours.

Chaque groupe assiste à un certain nombre de cours et chaque salle a une capacité limitée. Une solution est composée d'une liste ordonnée de longueur |C| où les positions correspondent aux cours (l'indice i correspond au cours i pour i = 1, …, |C|). Un nombre entier compris entre 1 et 36 (représentant un intervalle de temps) en position i indique l'intervalle de temps auquel l'événement i est affecté. Par exemple la liste 4 25 12 34 signifie que le cours 1 est affecté au créneau horaire 4, le cours 2 est affecté au créneau horaire 25, le cours 3 est affecté au créneau horaire 12 et le cours 4 est affecté au créneau horaire 34.

L'affectation des salles ne fait pas partie de la représentation explicite ; à la place, on utilise un algorithme d'appariement (couplage) pour les générer.

Pour chaque créneau horaire il y a une liste de cours y ayant place, et une liste des salles possibles auxquelles ces cours peuvent être affectés. L'algorithme d'appariement donne une correspondance de cardinalité maximale entre ces deux ensembles (listes) en utilisant un algorithme déterministe de flots dans les réseaux. S'il reste encore des cours non-placés, l'algorithme les prend dans l'ordre de l'étiquette et met chacun dans la salle qui satisfait les critères (type et capacité) et qui, à ce moment, n'est affectée qu'à un nombre minimum de cours. Si deux ou plusieurs salles satisfont les critères, il faut prendre celle avec la plus petite étiquette.

De cette façon, chaque affectation cours-créneau horaire correspond uniquement à un emploi du temps, c'est-à-dire une affectation complète des créneaux horaires et des salles à tous les cours.

La représentation de la solution décrite ci-dessus permet de définir une structure de voisinage à l'aide de mouvements simples impliquant uniquement les créneaux horaires et les cours. L'attribution des salles est prise en charge par l'algorithme d'appariement.

On a considéré trois types de mouvement sur un emploi du temps impliquant respectivement un, deux et trois cours :

- Un mouvement de type 1 déplace un cours à partir d'un créneau horaire vers un autre ;
- Un mouvement de type 2 échange deux cours dans deux créneaux horaires différents ;
- Un mouvement de type 3 permute trois cours dans trois créneaux horaires distincts dans une des deux manières possibles.

Ces trois mouvements sont décrits par des pseudo-algorithmes dans la méthode de résolution basée sur le recuit simulé (RS) que nous allons voir par la suite.
Chaque type de mouvement définit un voisinage noté $V1, V2, V3$.
Le voisinage complet V est alors défini comme l'union de ces trois quartiers : $V = V1 \cup V2 \cup V3$.

Lorsque le voisinage est grand, on utilise une première amélioration de la recherche locale qui fonctionne comme suit :

- On considère les mouvements de type 1, 2 et 3 (en ordre) pour chaque cours impliqué dans des violations de contraintes.
- Initialement, on ignore la violation des contraintes souples. Ensuite, si l'emploi du temps est faisable, on prend aussi bien les contraintes souples de telle sorte qu'on ne revienne jamais à une solution non-faisable.
- Pour chaque déplacement potentiel, on réapplique l'algorithme d'appariement au créneau horaire affecté.
- La recherche locale est terminée si aucune amélioration ne se trouve après avoir essayé tous les déplacements possibles sur chaque cours de l'EDT.

L'algorithme suivant décrit en détail la résolution :

1. $Cpt - cours := 0$;
   $Générer\ une\ liste\ circulaire\ de\ cours\ ordonnée\ aléatoirement$;
   $Initialiser\ un\ pointeur$ à $gauche\ du\ premier\ cours\ de\ la\ liste$;
2. $Déplacer\ le\ pointeur\ vers\ le\ cours\ suivant$;
   $Cpt - cours := Cpt - cours + 1$;
   $Si\ (Cpt - cours = |C|)\{$
   $Cpt - cours := 0$;
   $Aller\ vers\ 3$;
   }
   (a) $Si\ (cours\ courant\ ne\ provient\ pas\ la\ violation\ d'une\ contrainte\ hard)\{$
   $Aller\ vers\ 2$;
   }
   (b) $Si\ (il\ n'existe\ pas\ un\ mouvement\ pour\ ce\ cours)\{$

Aller vers 2 ;
}
(c) *Calculer le mouvement suivant* (*En* $V1, V2, V3$ *respectivement*)*et Générer l'EDT potentiel résulté* ;
(d) *Appliquer l'algorithme d'appariement sur le créneau horaire affecté par le déplacement* ;
(e) *Si* (*déplacement réduit la violation des contraintes hard*){
Déplacer ;
$Cpt - cours := 0$;
Aller vers 2 ;
}
(f) *Sinon*{
Aller vers 2.(*b*) ;
}
3. *Si* (*il reste une violation de contraintes hard*){
Terminer la recherche locale ;
}
4. *Déplacer le pointeur vers le cours suivant* ;
$Cpt - cours := Cpt - cours + 1$;
Si $(Cpt - cours = |C|)$ {
Terminer la recherche locale ;
}
(a) *Si* (*cours courant ne viole aucune contrainte soft*){
Aller vers 4 ;
}
(b) *Si* (*il n'existe pas un mouvement pour ce cours*){
Aller vers 4 ;
}
(c) *Calculer le mouvement suivant* (*En* $V1, V2, V3$ *respectivement*)*et Générer l'EDT potentiel résulté* ;
(d) *Appliquer l'algorithme d'appariement sur le créneau horaire affecté par le déplacement* ;
(e) *Si* (*déplacement réduit la violation des contraintes hard*){
Déplacer ;
$Cpt - cours = Cpt - cours + 1$;
Aller vers 4 ;
}
(f) *Sinon*{
Aller vers 4.(*b*) ;
}

Alg 1 Résolution du problème d'EDT par la recherche locale

b. Recuit simulé

Définition *(Wikipedia, Recuit simulé, 2016)*

Le recuit simulé (RS) (en anglais SA : Simulated Annealing) est une méthode empirique (métaheuristique) inspirée d'un processus utilisé en métallurgie. On alterne dans cette dernière des cycles de refroidissement lent et de réchauffage (*recuit*) qui ont pour effet de minimiser l'énergie du matériau. Cette méthode est transposée en optimisation pour trouver les extrema d'une fonction.
Elle a été mise au point par trois chercheurs de la société IBM, S. Kirkpatrick, C.D. Gelatt et M.P. Vecchi en 1983, et indépendamment par V. Černy en 1985.

La méthode vient du constat que le refroidissement naturel de certains métaux ne permet pas aux atomes de se placer dans la configuration la plus solide. La configuration la plus stable est atteinte en maîtrisant le refroidissement et en le ralentissant par un apport de chaleur externe, ou bien par une isolation.

Algorithme

Cet algorithme a été proposé par (E. Aycan, T. Ayav)**.**

L'application de recuit simulé au problème d'EDT relativement simple. Les particules sont remplacées par des éléments. L'énergie du système peut être définie par le coût d'emploi du temps pour la modélisation. Une allocation initiale est effectuée sur des éléments étant placés de manière aléatoire dans une période choisie. Le coût initial et la température initiale sont calculés. Pour déterminer la qualité de la solution, le coût joue un rôle essentiel dans l'algorithme et de même le rôle de l'énergie du système dans la qualité d'une particule étant recuite.

La température est utilisée pour commander la probabilité de l'augmentation du coût et peut être assimilé par la température des particules physiques.

La variation du coût est la différence de deux coûts ; un d'eux est le premier coût qui est avant la perturbation et le second est le coût après que l'élément aléatoirement choisi ait changé d'une activité. L'élément est déplacé si le changement du coût est accepté, ou bien qu'elle réduit le coût du système, ou l'augmentation est autorisée à la température actuelle.

Selon le modèle du problème d'EDT, le coût d'élimination d'un élément se compose généralement d'un coût de groupe, un coût d'enseignant et un coût de salle.

Le recuit simulé est une méthode itérative et un algorithme de type SA accepte une nouvelle solution si son coût est inférieur au coût de la solution courante à chaque itération. Même si le coût de la nouvelle solution est plus grand, il existe une probabilité que cette solution soit acceptée. Avec ce critère d'acceptation, il est alors possible de sortir une solution locale minimale. L'algorithme de SA utilisé, noté avec SA (P), est représenté dans (Alg 4) :

$Trouver\ une\ solution\ initiale\ aléatoire$;
$Sélectionner\ une\ température\ initiale\ t := t0 > 0$;
$Sélectionner\ une\ fonction\ de\ réduction\ de\ température\ \alpha$;

```
répéter{
    répéter{
        s' := chercherVoisin(s);
        δ := F(s') − F(s) ;
        Si((δ ≤ 0) ou (exp(−δ/t) < rand[0,1])){
            s := s';
        }
    } jusqu'à (cpt = nrep) ;
    t := Refroidir(t) ;
} jusqu'à (condition est vraie) ;
```

Alg 2 Résolution du probème d'EDT par le recuit simulé

Comme dans le cas de la recherche locale, une structure de voisinage doit être définie afin de mettre en œuvre l'algorithme du SA.

Dans ce travail, trois algorithmes sont traités en différentes combinaisons. Dans chaque itération de l'algorithme, la recherche de voisinage est effectuée une fois pour trouver la prochaine série de solution possible. Les algorithmes utilisés sont :

1. Recherche simple de voisinage (SSN) :
   Choisir au hasard un cours et un créneau horaire. Le créneau horaire choisi est affecté comme l'heure de début de l'activité sélectionné. Il est à noter que créneau-horaire(cr) représente le créneau horaire à partir du cours cr.

```
SSN(){
    cours := selectionner aléatoirement un cours ;
    ch := selectionner aléatoirement un créneau horaire;
    craineau − horaire(cours) := ch ;
}
```

Alg 3 Recherche simple de voisinage

2. Permutation voisine (SWN) :
   Sélectionner deux cours et permuter leurs heures de début.

```
SWN(){
    cours1 := selectionner aléatoirement un cours1 ;
    cours2 := sélectionner aléatoirement un cours2 ;
    ch := créneau − horaire(cours1) ;
    créneau − horaire(cours1) := créneau − horaire(cours2) ;
    créneau − horaire(cours2) := ch ;
}
```

Alg 4 Sélection et de permutation des voisins

3. Recherche et permutation simples de voisinage (S^3WN) :
   Cet algorithme choisit aléatoirement deux cours et deux créneaux horaires. Ces deux créneaux horaires sont attribués comme les heures de début des deux cours.

```
S3WN(){
        cours1 := selectionner aléatoirement un cours1 ;
        cours2 := selectionner aléatoirement un cours2 ;
        ch1 := selectionner aléatoirement un créneau horaire ;
        ch2 := selectionner aléatoirement un créneau horaire ;
        créneau − horaire1(cours1) := ch1;
        créneau − horaire1(cours1): = ch2;
}
```

Alg 5 Recherche et permutaion simples des voisins

Calcul du coût :

Le calcul du coût tente de montrer les influences des deux types contraintes (hard et soft). Chaque contrainte est définie par une fonction de « score ». Les contraintes dures sont représentées comme suit :

1. Si les créneaux des séances sont durs et violent les contraintes dures de cette séance :

$$Fc1 = w1 \sum_{i=1}^{n} CHi$$

   $\boldsymbol{n}$ est le nombre de séances, $\boldsymbol{w1}$ est le poids et $\boldsymbol{CHi}$ représente le nombre de créneaux horaires qui sont interdits aux séances.

2. Si le même enseignant est affecté à deux séances en même temps :

$$\mathrm{Fc2} \;=\; \mathrm{w2} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Eij$$

   $\boldsymbol{n}$ est le nombre de séances, $\boldsymbol{w2}$ est le poids, $\boldsymbol{Eij}$ est le nombre d'enseignants qui donnent deux cours, $\boldsymbol{i}$ et $\boldsymbol{j}$, en même temps.

3. Si la même promotion est affectée à deux séances en même temps :

$$\mathrm{Fc3} \;=\; \mathrm{w3} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pij$$

   $\boldsymbol{n}$ est le nombre de séances, $\boldsymbol{w3}$ est le poids, $\boldsymbol{Pij}$ est le nombre de promotions qui sont affectées à deux conférences, $\boldsymbol{i}$ et $\boldsymbol{j}$, en même temps.

4. Si les créneaux de séance sont séparés en deux jours (Chaque activité doit commencer et finir dans la même journée) :

$$Fc4 = w4 \sum_{i=1}^{n} Xi$$

   $\boldsymbol{n}$ est le nombre de séances, $\boldsymbol{Xi}$ est le nombre de créneaux horaires qui sont affectés aux cours $\boldsymbol{i}$, il est variable booléenne qui est vraie quand le cours est séparé en deux jours et $\boldsymbol{W4}$ est le poids.

5. Si les créneaux de séance sont des souples violent les contraintes souples de cette séance :

$$Fc5 = w5 \sum_{i=1}^{n} Yi$$

$\boldsymbol{n}$ est le nombre de séances, $\boldsymbol{Yi}$ est le nombre créneaux horaires qui dépend des préférences des enseignants et $\boldsymbol{w5}$ est le poids.

6. En cas de conflit des étudiants entre les cours échoués qu'un étudiant doit prendre et les cours réguliers qui doivent encore être pris.

$$\text{Fc6} = \text{w6} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} ETij$$

$\boldsymbol{n}$ est le nombre de séances, est le nombre $\boldsymbol{ETij}$ des étudiants qui prennent deux cours de différentes promotions, $\boldsymbol{i}$ et $\boldsymbol{j}$, en même temps. Si un étudiant suit un programme irrégulier, les conflits de cours sont minimisés par cette contrainte. Il est considéré comme une contrainte souple, sinon le problème n'aura pas de solution.

La fonction du coût est calculée comme la somme de ces contraintes dures et les contraintes souples, à savoir

$$F = FC1 + FC2 + FC3 + FC4 + FC5 + FC6$$

Refroidissement

Dans chaque itération N_{REP}, la température suivante est trouvée par la relation suivante :

$$t := \alpha t$$

où α est le paramètre de diminution pour le refroidissement géométrique et calculé par la relation suivante :

$\alpha = 1 - (ln(t) - ln(tf)) / Ndéplacement$

t est la température actuelle, tf est la température finale $et\ Ndéplacement$ est une valeur fixe qui affecte la durée de diminution de la température. Le paramètre de $nrep$ prend la valeur 3, qui renvoie le meilleur coût de la solution dans un temps d'exécution acceptable. Pour déterminer n_{rep}, plusieurs valeurs différentes telles que 1, 2, 3, 5, 6 et 10 sont expérimentés. Une température initiale t'_0 est affecté à 10000. Cette température est chaude assez pour permettre de se déplacer vers presque tous les états de voisinage, et l'algorithme SA met à jour itérativement la température à l'aide de la dépendance fonctionnelle entre l'acceptation de départ $\chi 0$ dont la probabilité (60% à 70%) et la température $T0$ de départ. Cette dépendance fonctionnelle est la suivante :

$X0 = X(\delta 1, \delta n, \delta n + 1, \cdots, \delta m, t0) = 1 / \sum_{i=1}^{n} 1\ exp(-\delta i / t0) + (m - n) / \text{n}$ où $\delta i = F(si) - F(s0)$, s0 est la solution initiale, δi est une solution voisine de s0, F est la fonction de coût, m est la taille de l'espace de solution voisin. t0 est la température initiale qui est dérivée à partir de la probabilité d'acceptation χ0 en utilisant l'algorithme présenté dans [4]. Par exemple, dans les paramètres dépendus dans ce travail, t0 est calculé comme 5000. Cet algorithme doit être exécuté qu'une seule fois avant d'exécuter l'algorithme du recuit simulé (SA).

4.1-2. Approches basées sur l'intelligence artificielle (IA)

L'intelligence artificielle s'intéresse à tous les problèmes dont la résolution ne peut être ramenée à une méthode simple, précise et algorithmique.

Dans l'IA, plusieurs approches sont utilisées pour la résolution de l'EDT. Elles sont classées sous trois catégories : les méthodes de construction, les méthodes de réparation et les méthodes d'évolution.

La figure montre les approches de chaque catégorie.

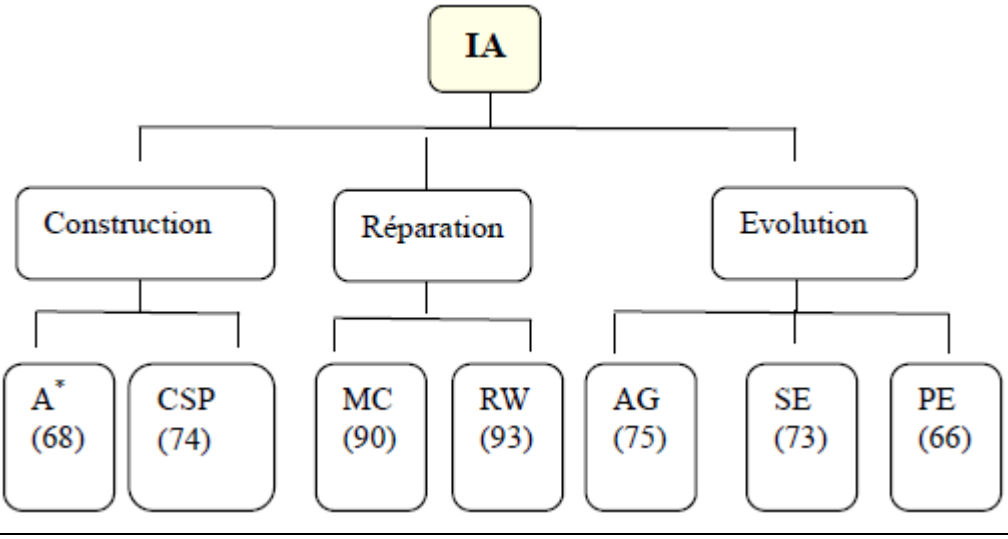

fig 2 Liste des méthodes d'intelligence artificielle de résolution du problème d'EDT (Houria, 2012)

Les méthodes de construction comprennent : les problèmes de satisfaction des contraintes (CSP) et l'algorithme A étoile (A*).

La deuxième classe (les méthodes de réparation) comprend : les heuristiques mini-conflit (MC) et les heuristiques random walk (RW).

La troisième classe (les méthodes d'évolution) contient : les algorithmes génétiques, les stratégies d'évolution et la programmation évolutive.

Nous allons présenter comme travail lié à la résolution du problème d'EDT dans le cadre de l'IA une seule approche, celle qui basée sur les AG. Et nous présenterons les CSP dans le chapitre 3 vue que c'est l'approche choisie dans notre étude.

Algorithmes génétiques (Wikipedia, Algorithme génétique, 2016)

Les algorithmes génétiques appartiennent à la famille des algorithmes évolutionnistes. Leur but est d'obtenir une solution approchée à un problème d'optimisation, lorsqu'il n'existe pas de méthode exacte (ou que la solution est inconnue) pour le résoudre en un temps raisonnable. Les algorithmes génétiques utilisent la notion de sélection naturelle et l'appliquent à une population de solutions potentielles au problème donné. La solution est approchée par « bonds » successifs, comme dans une procédure de séparation et évaluation (Branch-And-Bound), à ceci près que ce sont des formules qui sont recherchées et non plus directement des valeurs.

Algorithmes génétiques et le problème d'emploi du temps

Cette résolution est faite par **Branimir Sigl (et al)** (Branimir Sigl, Marin Golub, Vedran Mornar)

Initialement, la résolution du problème de génération automatique d'un emploi du temps dépend essentiellement sur le fait de définir une matrice binaire (comme l'on a déjà vu dans 2-2).

En prenant en considération toutes les contraintes posées dans ce problème, on trouve qu'il y a une sorte de redondance faisant agrandir la taille du problème. C'est pour cela qu'on a tenté d'éliminer du jeu quelques tierces. On parle ici des instructeurs (enseignants) et des groupes. Ce sont deux parties intégrantes de la définition d'une séance. Ces deux variables ne sont pas jetées, elles sont plutôt stockées pour que l'on puisse utiliser afin de résoudre les conflits.

En outre, seules les combinaisons possibles entre les séances, les locaux, les jours et les périodes sont générées.

Le pseudocode ci-dessous décrit le processus de génération des variables.

```
Pour toute séance s {
Générer toutes les combinaisons possibles des locaux
        Pour toute pair (jour, période) possible{
                Pour tout l dans les combinaisons des locaux{
                        Créer variable Xsjpl
                }
        }
}
```

Alg 6 Génération des variables de la matrice

Pour faciliter la génération d'un conflit, trois structures 3D auxiliaires sont créées.

Chaque structure représente une sorte de vue sur l'emploi du temps : vue de l'enseignant, vu du local et vue du groupe. Au cours du processus de génération de contraintes, chaque variable est positionnée, pour toutes les pairs (jour, période) possibles, à la coordonnée z appropriée qui indique les enseignants, les groupes ou les locaux alloués pour la séance correspondante. Après le remplissage de toutes les données, chaque coordonnée x-y-z est vérifiée. Si plusieurs séances se trouvent en compétition, une nouvelle contrainte doit être générée. Le solveur assure maintenant que seule une variable prend un local particulier dans un créneau temporel unique. On peut représenter cette contrainte par la relation suivante :

$\sum \boldsymbol{Xséance} \leq \mathbf{1}$. La figure ci-dessous est un exemple illustratif de cette formule.

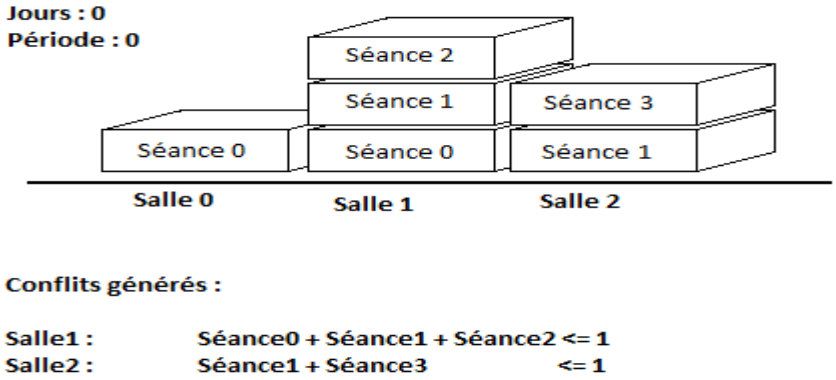

Fig 3 Génération des conflits

Ainsi, tous les conflits seront considérés et donc il ne restera plus qu'améliorer le résultat, chose que nous n'allons pas aborder vue la complexité des techniques qui se chargent de ce fait.

4.2- Approches distribuées : SMA et DCSP

La notion de distribution en informatique s'est apparue dans les années 70 dans les systèmes éducatifs (comme le réseau Cyclades en France 1971) dans le but de mettre en commun les ressources informatiques des centres universitaires. Et depuis, elle a connu une évolution très impressionnante, son utilisation s'est parallèlement propagée avec le progrès des différents systèmes informatiques.

L'apparition des systèmes multi-agents a effectivement marqué un point dans le développement des architectures distribuées. On parle maintenant de la distribution de l'intelligence.

Sur la base de profiter au maximum du partage des ressources et de distribuer les tâches tout en satisfaisant les besoins intervient l'idée de distribution dans le problème d'emploi du temps universitaire. Cependant, cette idée n'a pas vraiment atteint une part considérable sur ce genre de problèmes. Les problèmes de satisfaction distribuée des contraintes (DCSP) sont moins supportés dans la communauté scientifique que ceux centralisés.

Nous allons, après avoir pris des connaissances sur les systèmes multi-agents et les problèmes de satisfaction distribuée des contraintes, présenter une méthode proposée dans ce cadre à la résolution du problème d'emploi du temps universitaire.

4.2-1. Les systèmes multi-agents (SMA)

Le domaine des systèmes multi-agents est vaste et les recherches relatives à ce domaine sont de plus en plus progressées. Nous allons juste y prendre une vue globale qui nous aidera à comprendre le travail relatif à la résolution du problème d'EDT. Pour cela, nous devons d'abord savoir ce que signifie l'élément essentiel de ces systèmes (l'agent).

4.2-1.1. Concept d'agent

a. Définition (Ferber, 1995)

Un agent est une entité autonome, réelle ou abstraite, qui est capable d'agir sur elle-même et son environnement, qui, dans un univers multi-agents, peut communiquer avec d'autres agents, et

dont le comportement est la conséquence de ses observations, de ses connaissances et de ses interactions avec les autres agents.

b. Caractéristiques d'agents

Contrairement aux objets, les agents sont des entités autonomes, c.-à-d. qu'on ne peut pas obliger un agent d'exécuter une méthode comme le cas de l'objet ; il exécute plutôt ses comportements spécifiques en réponse à son désire.

La rationalité : Un agent rationnel est un agent qui est capable d'atteindre ses objectifs de façon optimale.

La situation dans un environnement : Un agent est généralement situé au moins dans un environnement et interagit avec.

La coopération : Les agents peuvent être coopératifs ; c.-à-d. qu'ils collaborent entre eux via différents mécanismes (comme les rendez-vous). Dans une situation de coopération, les agents relâcheraient leurs objectifs locaux afin d'atteindre l'objectif global du système.

La communicabilité : Généralement, les agents communiquent entre eux par les messages et avec leur environnement via les captures des traces.

L'adaptabilité : Les agents peuvent s'adapter avec dans leurs systèmes par l'apprentissage.

La flexibilité : Un agent flexible est un agent proactif (capable de prendre l'initiative et déclencher donc le système sans l'intervention d'une autre entité), réactif (capable de réagir après la réception d'un état du système) et sociable (il peut former une société avec les autres agents via la communication).

Capable de répondre à temps : l'agent doit être capable de percevoir son environnement et d'élaborer une réponse dans le temps requis.

4.2-1.2. Les systèmes multi-agents (SMA)

a. Définition (Houssem Eddine Nouri, Olfa Belkahla, 2014)

Selon **Y. Demazeau** un S.M.A est basé sur une décomposition en quatre parties :

- **Agents A**, qui concernent les modèles ou architectures utilisées pour la partie active de l'agent, depuis un simple automate jusqu'à un système complexe à base de connaissances.
- **Environnement E**, qui sont des milieux dans lesquels évoluent les agents. Ils sont généralement spatiaux.
- **Interactions I**, qui englobent les infrastructures, les langues et les protocoles d'interactions entre agents, depuis les interactions physiques jusqu'aux interactions par actes de langage.
- **Organisation O**, qui structurent les agents en groupes, hiérarchies, relations...

b. Types des systèmes multi-agents

Les SMA se diffèrent selon différents critères. Selon l'interaction on trouve (Houssem Eddine Nouri, Olfa Belkahla, 2014) :

- **SMA Ouverts :** Dans ce genre de S.M.A, aucune barrière n'est imposée aux agents (concernant leur migration). En effet, les agents peuvent entrer, ressortir et changer d'environnement. Ce type de S.M.A est en général utilisé pour les systèmes nécessitant des ressources (physiques ou virtuelles) qui ne peuvent être que dans un autre système. Ex : E-commerce.
- **SMA Fermés :** Aucun échange n'est autorisé, les agents ne doivent interagir qu'au sein de leur environnement et doivent se contenter de ce qui y est présent.

Si par contre, nous considérons les S.M.A selon le type d'agents les constituant :

- **SMA Hétérogènes :** Différents types d'agents peuvent se trouver au sein du même S.M.A. Les agents sont créés sur des modèles différents, mais arrivent à coexister ensemble dans le même environnement grâce aux standards de communication.
- **Homogènes :** Le système multi-agents est composé de plusieurs agents du même type.

c. Domaines d'application des SMA

Avec leur évolution remarquable, l'utilisation des systèmes multi-agents couvre aujourd'hui de plus en plus différents domaines tels que l'industrie, la santé, l'information, … etc. Nous allons citer quelques cas réels d'application où ils ont énormément réussi.

Apprentissage électronique (E-learning)

Les systèmes d'apprentissage en ligne ont un fort besoin d'adaptation de leur contenu et des activités pédagogiques qu'ils proposent en fonction du niveau de connaissances de l'utilisateur. (LEMOUZY, 2011)

Commerce électronique (E-commerce)

Beaucoup d'entreprises sont aujourd'hui en train de développer des applications agents pour le commerce électronique et les utiliser à Internet.

La nouvelle génération d'elles permettent les utilisateurs (les consommateurs et les vendeurs) la possibilité de négocier pour un produit entre eux : les sites pour les ventes aux enchères, pour les négociations entre les utilisateurs, etc.

Il y a des modèles créés par le professeur J-P.**SANSONNET** (J-P.SANSONNET, 2005) qui

Recherche d'information

Les systèmes de recherche d'information ont pour but de diffuser un pool d'informations hypermédia. Les encyclopédies multimédia ou les guides virtuels font par exemple partie de cette classe. Les Système Hypermédia Adaptatifs **(**SHA) permettent de proposer l'information la plus en adéquation avec les intérêts et/ou les objectifs des utilisateurs. (LEMOUZY, 2011)

4.2-2. Les problèmes de satisfaction des contraintes distribuées (DCSP)

Un DCSP est un CSP dans lequel les variables et les contraintes sont distribuées entre des agents. Ces Contraintes se divisent en deux ensembles disjoints : les contraintes intra-agents et les contraintes inter-agents.

Une contrainte intra-agent n'est connue que par un agent. Habituellement, on considère qu'une contrainte inter-agent est connue par tous les agents ayant une variable dans cette contrainte. Comme dans le cas centralisé, résoudre un DCSP consiste à trouver une assignation de valeur aux variables en ne violant aucune contrainte (bien que la littérature des DCSP se concentre principalement à la résolution des contraintes inter-agents).
Trouver une assignation de valeur aux variables inter-agents peut être vu comme réaliser la cohérence ou la consistance d'un système multi-agent.
Les heuristiques de résolution d'un DCSP se classent en deux catégories. D'une part, la **programmation orientée-marché repose sur les mécanismes d'enchères**. D'autre part, des agents peuvent être en **compétition** pour utiliser des ressources, ces ressources pouvant calculer leur demande. En outre, les algorithmes de résolution des CSP semblent similaires à des méthodes de traitement parallèle/distribué pour résoudre des CSP, quoique les motivations de recherche soient fondamentalement différentes. (Thierry Moyaux, Brahim Chaib-draa, Sophie D'Amours)

4.2-3. SMA, DCSP et emploi du temps

Une approche proposée par (Thierry Moyaux, Brahim Chaib-draa, Sophie D'Amours) est basée sur le retour en arrière (backtrack) des DCSP pour obtenir l'emploi du temps recherché de façon distribué.

Nous allons voir comment en est formulé ce DCSP ainsi l'algorithme proposé.

a. Formulation du DCSP

Le problème a été formalisé par l'affectation des enseignants et des salles à des cours tout en respectant des créneaux horaires prédéfinis.

On a défini au préalable le nombre de jours et le nombre de cours par jours. Ensuite, on a supposé que les enseignants ne sont pas toujours disponibles, il fallait donc préciser les jours et les créneaux horaires de disponibilité pour chaque enseignant. On a aussi un nombre de cours hebdomadaire limité dont il est interdit de dépasser. Enfin, chaque cours concerne un ou quelques enseignants et pas tous, il fallait spécifier pour chaque enseignant les cours dont il puisse se charger.

On a défini le problème de génération d'emploi du temps par cinq ensembles :

$$\boldsymbol{Problème} = \{\boldsymbol{Enseignants}, \boldsymbol{Salles}, \boldsymbol{Cours}, \boldsymbol{Horaires}, \boldsymbol{Contraintes}\}$$

Où :

- ✓ **Enseignants** = $\{E_1, E_2, \ldots, E_e\}$ est l'ensemble des e professeurs ;
- ✓ **Salles** = $\{S_1, S_2, \ldots, S_s\}$ est l'ensemble des s salles ;
- ✓ **Cours** = $\{C_1, C_2, \ldots, C_c\}$ est l'ensemble des c cours à donner durant la session considérée ;
- ✓ **Horaires** = $\{H_1, H_2, \ldots, H_h\}$ est l'ensemble des h créneaux horaires disponibles dans une semaine ;
- ✓ **Contraintes** est l'ensemble des contraintes entre les variables des quatre ensembles précédents.

Comme on a vu que le problème d'emploi du temps universitaire peut être défini par une matrice binaire, un problème se pose lorsque la notion de distribution entre en jeux : l'accès concurrentiel

aux variables critiques (les locaux dans ce cas). C'est pour cela qu'ils ont proposé de définir pour chaque agent une matrice qui ne porte que les informations auquel sont relatives. On doit définir trois types de matrices :

$XEi[Sj][Hk][Cl] = X[E_i][S_j][H_k][C_l]$ **:** L'enseignant E_i ne connait que le contenu XEi extrait de la matrice X (le contenu qui le concerne).
$XCl[Ei][Sj][Hk] = X[Ei][Sj][Hk][Cl]$: Le cours Cl connait le contenu de XCl.
$XSj[Pi][Hk][Cl] = X[Ei][Sj][Hk][Cl]$: La salle Sj connait le contenu de XSj

b. Algorithme

Cet algorithme a été proposé par (Thierry Moyaux, Brahim Chaib-draa, Sophie D'Amours)**.** En raison qu'un enseignant puisse être prioritaire aux autres enseignants, et que certains cours n'étant affectés qu'à cet enseignant puissent être négligés, on a évité que les agents-enseignants se charger de la résolution du DCSP. Par contre, en choisissant les agents-cours comme les résolveurs du DCSP, aucun cours ne sera évitable. Même s'il y a des enseignants qui ne sont pas pris, c'est un problème moins difficile que ce des cours, vu que l'obligation est de planifier tous les cours prévus.

La figure ci-dessous décrit la philosophie suivie par les agents pour s'aboutir à la génération d'emploi du temps.

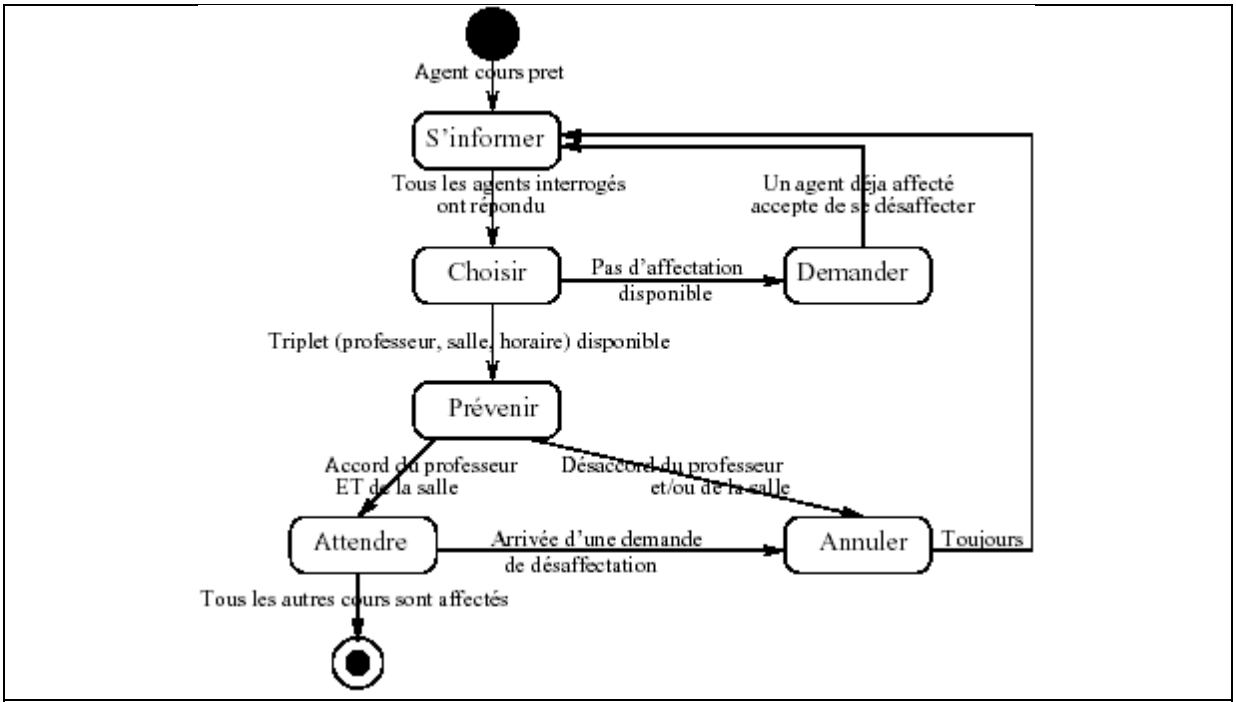

fig 4 Génération d'EDT par les agents cours

Pour chaque agent C_l, il s'agit de « noircir » (mettre à la valeur « vrai ») des cases de la matrice X_{Cl} en fonction de la négociation avec les autres agents-cours et des données provenant des agents-professeur et -salle, pour finalement y choisir une case « blanche » (dont la valeur est « faux »). Chaque agent-cours C_l commence par récolter les données des agents-professeur et -salle pour mettre à jour sa matrice X_{Cl} (état « S'informer »), puis il cherche dans cette matrice une affectation {professeur, salle, horaire} libre (état « Choisir »). S'il n'en trouve pas, il demande à un autre agent-cours qui a lui-aussi trouvé une affectation de renoncer à cette affectation (état «

Demander »), puis recommence l'algorithme au début. Si au contraire il trouve une affectation libre, il informe les agents-professeur et -salle concernés de l'horaire qu'il aimerait les retenir (état « Prévenir »). Si l'un de ces deux agents lui répond (ou les deux) qu'il ne peut pas, l'agent C_l prévient l'autre agent qu'il ne veut plus de cet horaire (état « Annuler ») et recommence l'algorithme au début. Si au contraire ces deux agents sont d'accord, notre agent C_l note que cette affectation est la sienne, puis il informe les autres agents-cours qu'il s'est trouvé une affectation et se met enfin en attente d'un message provenant d'un autre agent-cours qui lui demanderait de se désaffecter (état « Attendre »). Une fois que tous les agents cours ont averti l'agent C_l qu'ils ont eux-aussi trouvé leur affectation, celui-ci termine son exécution (état final).

5. Conclusion

Nous avons vu dans ce chapitre le problème d'emploi du temps selon différentes dimensions : EDT vu comme problème de recherche et comme problème d'optimisation. Nous avons présenté ensuite quelques travaux liés à la résolution de ce problème basés sur différentes stratégies.

Un point de vue personnel, la stratégie idéale à résoudre le problème d'EDT serait une méthode distribuée (DCSP). Une telle stratégie satisfait au maximum les contraintes, car elle prend en considération toute sorte de conflits y compris ceux qui peuvent s'introduire dans le cas où il y a des tant de ressources partagées entre les départements, on parle ici précisément des amphithéâtres. Cependant, ces méthodes ne sont pas également développées aussi bien que les méthodes centralisées, elles sont encore jeunes. La résolution n'est faite qu'en comptant sur les systèmes multi-agents qui sont plus coûteux que les mécanismes se basant sur les paradigmes classiques (programmation orientée objet).

Pour notre cas, nous avons choisi de résoudre le problème d'emploi du temps universitaire en s'inspirant des problèmes de satisfaction des contraintes. Le progrès des méthodes de résolution des CSP ne cesse pas de continuer. Les outils qui se chargent de cette résolution sont de plus en plus améliorés, ils y offrent de multiples stratégies, ils réduisent énormément l'effort de programmation.

Avant de passer au traitement de notre cas, nous allons voir dans le chapitre suivant quelques notions sur la méthode et les outils dont nous allons dépendre dans la phase de résolution.

CHAPITRE 02 : OUTILS PRELIMINAIRES

1. Introduction

Ce chapitre est consacré aux notions fondamentales et nécessaires pour la résolution du problème d'emploi du temps. Nous allons commencer par une brève introduction à la programmation par contraintes (PPC). Dans la suite, nous allons nous intéresser aux mécanismes d'inférence qui ont été intégrés à la programmation logique (PL) pour aboutir à la programmation logique par contraintes (PLC). Nous finirons par présenter le langage PROLOG et ses éléments de base. Ce dernier est considéré comme l'un des solutions élégantes et efficaces pour les problèmes d'EDT.

2. Programmation par contraintes (PPC)

2.1. Définition (Wikipeida, 2016)

La programmation par contraintes (PPC, ou *CP* pour *Constraint Programming* en anglais) est un paradigme de programmation apparu dans les années 1980 permettant de résoudre des problèmes combinatoires de grandes tailles tels que les problèmes de planification et d'ordonnancement. En programmation par contraintes, on sépare la partie modélisation à l'aide de problèmes de satisfaction de contraintes (ou CSP pour *Constraint Satisfaction Problem*), de la partie résolution dont la particularité réside dans l'utilisation active des contraintes du problème pour réduire la taille de l'espace des solutions à parcourir (on parle de propagation de contraintes).

2.2. Problèmes de satisfaction des contraintes (CSP)

Un CSP peut être défini par un 3-uplet (V, D, C) tel que :

- V est un ensemble de *variables* ; $V = \{V1, V2, \ldots, Vn\}$;
- D est un ensemble de *domaines finis* ; $D = \{D1, D2, \ldots, Dn\}$ où chaque domaine Di est l'ensemble des valeurs de Vi ;
- C est un ensemble conjonctif de *contraintes* ; $C = \{C1, C2, \ldots, Cm\}$ où chaque contrainte Ci est définie par :
  - un sous-ensemble de variables $var(Ci) = \{Vi1, Vi2, \ldots, Vini\}$;
  - une relation $rel(Ci) = rel(Vi1, Vi2, \ldots, Vini) \subseteq Di1 \times Di2 \times \ldots \times Dini$..

Une solution est un n-uplet $\{v1, v2, \ldots, vn\}$ (Patrick ESQUIROL, Pierre LOPEZ, 1995) de valeurs (une par variable) tel que pour chaque contrainte k , les valeurs associées aux variables $var(Ck)$ assurent le respect de Ck. Sur ce modèle, on peut décliner plusieurs types de problèmes, de complexité équivalente (NP-complets), le problème de la satisfaction d'un ensemble de contraintes sur des variables à domaine fini pouvant se ramener à celui de la coloration d'une carte : celui qui consiste à colorier les régions d'une carte de façon à ce que toutes les régions ayant des frontières communes soient coloriées avec des couleurs différentes.

- démontrer l'existence de solutions, trouver une (toutes les) solution(s) ;
- démontrer qu'une instanciation partielle des variables caractérise au moins une (toutes les) solution(s) ;
- déterminer toutes les valeurs possibles d'une variable sur l'ensemble des solutions ;
- trouver une (toutes les) solution(s) optimale(s).

2.3. Notion de consistance dans les CSP et propagation par contraintes (Patrick ESQUIROL, Pierre LOPEZ, 1995)

La *consistance* est une propriété établie par comparaison entre les valeurs d'une ou plusieurs variables et les valeurs autorisées par les contraintes. Le retrait (filtrage) de valeurs ou de n-uplets de valeurs inconsistants constitue un *renforcement de cohérence* ou *propagation de contraintes*. Un filtrage complet permet en théorie d'obtenir une représentation explicite de l'ensemble des solutions. L'absence de solution, ou inconsistance globale, est détectée à l'issue d'un filtrage, s'il existe une variable Vi telle que $Di = \emptyset$.

Les algorithmes de filtrage ont pour objet de transformer un CSP en un nouveau CSP' tel qu'un certain type de consistance soit vérifié. Ils se distinguent selon l'arité des contraintes considérées :

- **Contraintes unaires**

Un CSP est *domaine-consistant* si le domaine de chaque variable est cohérent avec l'ensemble des contraintes unaires qui pèsent sur elle.

- **Contraintes binaires**

  - Un CSP est *arc-consistant* si le domaine de chaque variable est cohérent avec l'ensemble des contraintes binaires qui pèsent sur elle.
  - Un CSP est *chemin-consistant* si tout couple de valeurs autorisé par une contrainte liant 2 variables l'est aussi par tous les chemins de contraintes liant ces variables.

La chemin-consistance est une condition plus forte que l'arc-consistance mais ne constitue pas une condition suffisante de consistance globale.

- **Contraintes n-aires**

Un CSP est *k-consistant* si toute instanciation localement consistante de $k-1$ variables, peut être étendue à toute instanciation localement consistante de k variables.

En l'absence d'hypothèses particulières (*e.g.*, contraintes numériques binaires conjonctives), il n'existe pas d'algorithme polynomial pour vérifier la consistance globale d'un réseau de contraintes quelconques. Il n'est donc pas possible d'implanter un mécanisme général efficace dans un langage de PLC.

3. Programmation logique (PL)

3.1. Définition (Wikipedia, Programmation logique, 2015)

La programmation logique est une forme de programmation qui définit les applications à l'aide d'un ensemble de faits élémentaires les concernant et de règles de logique leur associant des conséquences plus ou moins directes. Ces faits et ces règles sont exploités par un démonstrateur de théorème ou moteur d'inférence, en réaction à une question ou requête.

Cette approche se révèle beaucoup plus souple que la définition d'une succession d'instructions que l'ordinateur exécuterait. La programmation logique est considérée comme une programmation déclarative plutôt qu'impérative, car elle s'attache davantage au *quoi* qu'au *comment*, le moteur assumant une large part des enchaînements. Elle est particulièrement adaptée aux besoins de l'intelligence artificielle, dont elle est un des principaux outils.

3.2. Langages de programmation logique

Plusieurs langages de programmation logique ont été fondés depuis les années 80. Nous allons nous fixer sur deux exemples de ces langages : PROLOG et CLIPS.

3.2-1. Prolog

PROLOG est l'un des fameux langages les plus adaptés au paradigme de la programmation logique. Il est resté toujours supporté par plusieurs communautés scientifiques. Par conséquent, la syntaxe est un peu différente d'une édition à une autre. Nous trouvons parmi ces éditions : SWI-PROLOG, SICSTus PROLOG, GNU-PROLOG, …

Nous laissons l'étude de ce langage jusqu'après avoir des connaissances sur la programmation logique par contraintes (PLC) vue que c'est le langage dont nous allons nous baser dans notre travail.

3.2-2. CLIPS

Définition (Wikipedia, Clips (langage), 2015)

CLIPS est l'acronyme de **C L**anguage **I**ntegrated **P**roduction **S**ystem. Ce langage a été élaboré par la NASA pour développer rapidement et à moindre coût des programmes portable sur différentes plates-formes et aisément intégrable avec d'autres applications. Le choix du langage C s'explique ainsi au vu de ces objectifs. A cet effet, le code source est disponible pour pouvoir utiliser CLIPS sur toute plate-forme disposant d'un compilateur C ANSI.
CLIPS est ce qu'on appelle un générateur de Systèmes Experts (*Shell* en anglais) à **base de règles** de production.
Il propose :

- un Moteur d'Inférence en chaînage avant ;
- un langage orienté objet (COOL) ;
- une programmation procédurale.

CLIPS est un Système Expert permettant de manipuler des faits et règles à l'aide d'un moteur d'inférence en chaînage avant. Nous précisons tout d'abord la syntaxe du langage, avant d'aborder la problématique de la représentation de connaissance en CLIPS.
CLIPS utilise une syntaxe similaire à LISP. Ceci se traduit par l'omniprésence de parenthèses ouvrantes "(" et fermantes ")". Elles sont notamment nécessaires pour appeler des fonctions systèmes ou définies par le programmeur (*programmation procédurale*).
Dans CLIPS, les opérations mathématiques utilisent une notation préfixée. Prenons un petit exemple pour cette notation : 3+2 et 2*5-8 s'écrivent en CLIPS comme suivant :

```
CLIPS > (+ 3 2)
5
CLIPS > (- (* 2 5) 8)
2
CLIPS >
```

Types de données

Type de donnée	Définition	Exemple
Integer	[+|-] nombre	1 +3 -1 65
Float	**Integer** [.[nombre] [e [+|-] nombre]	1.5 1.0 9e-1 3.5e1
Symbol	<lettre|chiffre|!|#|^|*><caractères>* **Attention :** CLIPS distingue minuscules et majuscules	Bonjour, bonjour, hello-world, 345B, 127-0-0-1
String	"><caractère>*<">	"hello world" ""\"hello\" " "10 francs "
External address	Adresse mémoire d'une structure de donnée externe i.e. retournée par une **fonction utilisateur**[1] <Pointer-XXXXXX> où XXXXXX est l'adresse mémoire externe	<Pointer-00CF61AB>
Instance name	<[> **Symbol** <]> : nom d'une instance de classe dans COOL	[pump-1]
Instance address	Adresse mémoire d'un objet COOL. <Instance-XXX> où XXX est le nom de l'instance.	[Instance-pump-1]

Fig 5 Types de données en CLIPS

Exemple

- Les hommes sont mortels
- Socrate est un homme
- Les chiens sont mortels

Se traduit en CLIPS comme suit :

$(deffacts\ vérités$
$\quad(est\ homme\ mortel)$
$\quad(est\ Socrate\ homme)$
$\quad(est\ chien\ mortel)$
$)$

Et la règle suivante :

- si **a** est **b** et **b** est **c**, alors **a** est **c**

Qui se traduit en CLIPS par :

$(defrule\ translation$
$\quad(est\ ?a\ ?b)$
$\quad(est\ ?b\ ?c)$
$\quad\Rightarrow$
$\quad(assert\ (est\ ?a\ ?c))$
$)$

Avant la première exécution la base de faits contient donc ceci :

$1-\ (est\ homme\ mortel)$
$2-\ (est\ Socrate\ homme)$
$3-\ (est\ chien\ mortel)$

En lançant la résolution, la règle est appliquée une fois avec les faits 1 et 2 :

- Socrate est un homme, or tout homme est mortel, donc Socrate est mortel.

La base de fait contient maintenant ceci :

1 − $(est\ homme\ mortel)$
2 − $(est\ Socrate\ homme)$
3 − $(est\ chien\ mortel)$
4 − $(est\ Socrate\ mortel)$

La règle ne peut plus être appliquée à aucun fait, l'exécution s'arrête. Nous remarquons qu'un seul nouveau fait a été introduit dans la base de faits (Socrate est mortel). En effet, on ne peut pas en déduire que *Socrate est un chien*, comme dans le fameux sophisme.

4. Programmation logique par contraintes (PLC) (CLP)

La programmation logique par contraintes est un outil de résoudre les problèmes de satisfaction des contraintes (CSP).

Pourquoi utiliser la programmation logique par contraintes ?

Un trait saillant des CSP combinatoires est que toutes les variables prennent des valeurs de domaines finis. Il en résulte que, en théorie, tout CSP peut être soit démontré qu'il n'a pas de solution ou qu'il est résolu en utilisant un algorithme de recherche exhaustive ou une approche d'énumération directe. Par conséquent, la sagesse de développer des outils spéciaux pour de tels problèmes peut être remise en question : pourquoi les outils actuels de résolution des CSP combinatoires sont meilleurs que la recherche exhaustive ?

1. En raison de l'efficacité numérique de la détermination des solutions du CSP, ce qui a un effet négatif dans le cas d'une recherche exhaustive ou dans les cas où le nombre des variables est élevé. (Exemple) : on considère 30 variables. Chacune d'elles peut prendre 100 valeurs différentes. Le nombre total des ensembles des solutions possible (l'espace de recherche) est 100^{30} (10^{60})
2. Les programmes de CLP sont déclaratifs. La déclarativité signifie qu'une description bien formalisée du problème résolu revient au programme qui le résout. En outre, la programmation avec les langages de CLP offre la résolution aux problèmes sans concevoir des algorithmes, dans les compilateurs de CLP sont intégrés différents algorithmes qui servent à cette tâche.

5. Prolog

Le langage PROLOG est basé sur le calcul des prédicats du premier ordre (avec quelques extensions et restrictions). C'est une implémentation du Principe de Résolution (Robinson 1965) avec des stratégies particulières et des restrictions.

5.1. Définition (Wikipedia, Prolog, 2016)

Prolog est l'un des principaux langages de programmation logique. Le nom *Prolog* est un acronyme de PROgrammation en LOGique. Il a été créé par Alain Colmerauer et Philippe Roussel vers 1972. Le but était de créer un langage de programmation où seraient définies les règles logiques attendues d'une solution et de laisser le compilateur la transformer en séquence d'instructions. L'un des gains attendus était une facilité accrue de maintenance des applications,

l'ajout ou la suppression de règles au cours du temps n'obligeant pas à réexaminer toutes les autres.

Prolog est utilisé en intelligence artificielle et dans le traitement linguistique par ordinateur (principalement langages naturels). Ses règles de syntaxe et sa sémantique sont simples et considérées comme claires (un des objectifs poursuivis était de procurer un outil aux linguistes ignorant l'informatique). Les premiers résultats obtenus avec Prolog suscitèrent quelque temps, dans les années 1980, des recherches sur une cinquième génération, matérielle et logicielle, d'ordinateurs (nommée *Cinquième génération japonaise* en raison de l'engagement important du MITI sur le projet). L'effort engagé fut important, les retombées plus modestes, Prolog restant juste un langage parmi d'autres dans la panoplie du programmeur.
Prolog est basé sur le calcul des prédicats du premier ordre ; cependant il est restreint dans sa version initiale à n'accepter que les clauses de Horn[1] (les versions modernes de Prolog acceptent des prédicats plus complexes, notamment avec le traitement de la négation par l'échec[2]). L'exécution d'un programme Prolog est effectivement une application du théorème prouvant par résolution du premier ordre. Les concepts fondamentaux sont l'unification, la récursivité et le retour sur trace.

On peut construire en Prolog une base de connaissances dans un ordre indéterminé, puisque seules comptent les relations en présence et non leur séquence d'écriture. Prolog peut ensuite résoudre des séries de problèmes logiques relatifs à une telle base de connaissances (notion base de données déductive), problème similaire à la recherche d'une issue (ou plusieurs) dans un labyrinthe de contraintes établies.

5.2. Eléments de base de PROLOG (HERITAGE, 2010)

Nous présentons ici quelques éléments de base du langage PROLOG, c.-à-d. la syntaxe, la sémantique de ce langage et quelques bibliothèques dont nous dépondons dans notre cas, celles qui concernent la programmation logique par contraintes afin de bien nous mettre en place à ce paradigme de programmation et de nous familiariser avec.

5.2-1. Termes

Atomes

Les textes constants constituent des *atomes*. Un atome est ordinairement constitué d'une chaîne de lettres, nombres et traits bas (_), commençant par une lettre *minuscule*. Pour introduire un atome non alphanumérique, on l'entoure d'apostrophes : ainsi '+' est un atome, + un opérateur).

Arbres

Toutes les données manipulées en Prolog sont des arbres éventuellement infinis. Ces arbres sont formés de nœuds étiquetés :

- soit par une constante et, dans ce cas, ils n'ont aucun fils,
- soit par le caractère "point" et, dans ce cas ils ont deux fils,

[1] Une clause de Horn est une clause comportant au plus un *littéral positif.*

[2] C'est une règle d'inférence en programmation logique, utilisée pour la dérivation de $not\ p$ à partir de l'échec de la dérivation de p. En Prolog, la négation par l'échec est habituellement implémentée en utilisant les fonctionnalités non logiques du langage.

- soit par " <> " ou" < −> " ou " < −−> " ou " < − − −> " ou… et, dans ce cas, le nombre de traits d'union correspond au nombre de leurs fils.

Variables

Les variables sont indiquées en utilisant un ensemble de lettres, nombres et caractères de soulignement et commençant avec une lettre *majuscule*.

Ainsi, *X3* comme *Prix_Unitaire* sont des noms de variables admissibles.

Contrairement aux langages de programmation impératifs, la variable en PROLOG n'est pas un contenant auquel on affecte une valeur, mais représente (comme en mathématiques dans *X > 0*) l'ensemble des valeurs admissibles pour elle dans le cadre des contraintes.

Le champ initialement indéfini de la variable se précise par l'unification autour des contraintes. Une fois la variable unifiée, sa valeur ne peut plus être modifiée au sein d'une même branche d'évaluation. Le retour sur trace permet toutefois : de revenir sur cette unification dans le cadre de la poursuite de la recherche de valeurs admissibles (ou de *nouvelles* valeurs admissibles), dans le cadre d'une exploration exhaustive.

La *variable anonyme* est écrite avec un tiret bas (_). Toute variable dont le nom commence par un tiret bas est également une variable anonyme. C'est, comme le *x* ou le *y* de l'algèbre, une variable muette servant d'intermédiaire de calcul.

Nombres

Il y a deux types de nombres en PROLOG : les entiers (integer) et les réels (float).

Chaines de caractères

Les chaînes de caractères sont encadrées par des doubles quotes « " ».

Listes

Une liste n'est pas un type de données isolé, mais est définie par une construction récursive (utilisant le foncteur d'arité 2, c'est donc au niveau de la représentation interne un terme composé) :

1. l'atome [ ] est une liste vide ;
2. si *T* est une liste et *H* est un élément, alors le terme (H, T) est une liste.

Le premier élément, appelé la tête, est *H*, suivi par les contenus du reste de la liste, indiqué comme *T* ou queue. La liste *[1, 2, 3]* serait représentée en interne comme $'(1,'.'(2,'.'(3,[]))'$. Un raccourci de syntaxe est $[H \mid T]$, lequel est surtout utilisé pour construire des règles. La totalité d'une liste peut être traitée en agissant sur le premier élément, et ensuite sur le reste de la liste, par récursivité, comme en LISP.

Pour la commodité du programmeur, les listes peuvent être construites et déconstruites de diverses manières.

- Énumération d'éléments : $[abc, 1, f(x), Y, g(A, rst)]$
- Extraction de l'élément de tête : $[abc \mid L1]$
- Extraction de plusieurs éléments de tête : $[abc, 1, f(x) \mid L2]$
- Expansion du terme : $'(abc,'.'(1,'.'(f(x),'.'(Y,'.'(g(A,rst),[])))))'$.

Termes composés

Prolog ne peut représenter des données complexes que par *termes composés*. Un terme composé consiste en une tête (aussi appelée foncteur), qui doit être un atome, et des paramètres sans restriction de type. Le nombre de paramètres, nommé arité du terme, est en revanche significatif. Un terme composé est identifié par sa tête et son arité, et habituellement écrit comme foncteur/arité.

Exemples de termes composés :

- enseigne(enseignant1, module1)
  Le foncteur est *enseigne* et l'arité 2, le terme composé s'écrit enseigne/2.
- $f(g(X), h(Y))$
  Le foncteur est *f* et l'arité 2, le terme composé s'écrit f/2.
- initialisation
  Le foncteur est *initialisation* et l'arité 0, le terme composé s'écrit initialisation/0. Un atome est donc un terme composé d'arité 0.

5.2-2. Prédicats

La programmation en Prolog est très différente de la programmation dans un langage impératif. En Prolog, on alimente une base de connaissances de faits et de règles ; il est alors possible de faire des requêtes à la base de connaissances.

L'unité de base de Prolog est le prédicat, qui est défini comme étant vrai. Un prédicat consiste en une tête et un nombre d'arguments. Par exemple :

$universite(oeb)$.

Ici 'universite' est la tête, et 'oeb' est l'argument. Voici quelques demandes simples qu'on peut demander à un interpréteur Prolog basé sur ce fait :

$?- universite(oeb)$.
  oui.
$?- universite(X)$.
  $X = oeb$;
  $fail$.

Dans ce second exemple, à la question 'universite(X)' l'interpréteur propose la réponse 'X = oeb' unifiant la variable 'X' à l'atome 'oeb'. En Prolog il s'agit d'un *succès*. Après cette première réponse, l'utilisateur peut demander s'il y a d'autres réponses en utilisant le « ; » (symbole de la disjonction), ici l'interpréteur répond qu'il n'en trouve pas. Cette recherche d'autres solutions repose sur un modèle d'exécution non-déterministe (au sens du non-déterminisme des automates non-déterministes) avec retour sur les différents points de choix et exploration des alternatives non explorées.

$?- universite(constantine2)$.
  $fail$.

Ici, à la question ' universite(constantine2)' l'interpréteur répond qu'il ne peut pas prouver ce fait, en Prolog il s'agit d'un *échec*. En faisant l'hypothèse que tous les faits sont connus, cela signifie que constantine2 n'est pas une université.

Les prédicats sont en général définis pour exprimer les faits que le programme connaît à propos du monde. Dans la plupart des cas, l'usage de prédicats requiert une certaine convention. Par exemple, la sémantique des deux prédicats suivants n'est pas immédiate :
$pere(maria, omar)$.
$pere(omar, maria)$.

Dans les deux cas, 'père' est la tête tandis que 'marie' et 'pierre' sont les arguments. Cependant, dans le premier cas, maria vient en premier dans la liste des arguments, et dans le second c'est omar (l'ordre dans la liste des arguments importe). Le premier cas est un exemple d'une définition dans l'ordre verbe-objet-sujet et pourrait se lire avec l'auxiliaire 'avoir' : maria a pour père omar, et le second de verbe-sujet-objet et pourrait se lire avec l'auxiliaire 'être' : omar est le père de maria. Comme Prolog ne comprend pas le langage naturel, les deux versions sont correctes en ce qui le concerne ; cependant il est important d'adopter des normes de programmation cohérentes pour un même programme. En général, c'est plutôt l'auxiliaire 'être' qui est utilisé.

On conviendra par exemple que $famille(omar, sara, [mohammed, bilel, maria])$. signifie que omar et sara sont respectivement le père et la mère de 3 enfants : mohammed, bilel et maria ; "famille" est alors un prédicat à 3 termes, le dernier étant une liste d'un nombre quelconque (éventuellement nul) d'enfants.

5.2-3. Règles

Un exemple de règle est :
$lumière(on) :- interrupteur(on)$.
Le « $:-$ » signifie « *si* » ; cette règle indique lumière(on) est vraie si interrupteur(on) est vrai.
Les règles peuvent aussi utiliser des variables comme :
$père(X, Y) :- parent(X, Y), mâle(X)$.
Pour signifier qu'un X est père d'un Y *si* X est parent de Y *et* X est mâle, où " , " indique une conjonction.
On pourrait avoir de même :
$parent(X, Y) :- père(X, Y) ; mère(X, Y)$.
Pour signifier qu'un X est parent d'un Y *si* X est père de Y *ou* X est mère de Y, où " ; " indique une alternative.

Un fait est un cas particulier de règle. En effet les deux lignes suivantes sont équivalentes :
a.
$a :- true$.

5.3. Fonctionnement de l'interpréteur (Pastre, 1999)

L'interpréteur prend le premier but b1, qu'il essaie de résoudre, c'est-à-dire il essaie d'unifier b1 avec une tête de clause. S'il réussit, il cherche à résoudre la queue de la clause, instanciée par l'unification, en résolvant, dans l'ordre, chacun des littéraux de cette queue ; puis il essaie de résoudre le prochain but en attente. En cas d'échec, il y a retour arrière au dernier choix effectué. Quand une solution a été trouvée, on peut, soit arrêter la recherche, en tapant [Entrée] (Retour-Charriot), soit demander les autres, en tapant ; suivi de [Entrée]. (Dans certains PROLOG, on

a systématiquement toutes les solutions.) La résolution est terminée s'il n'y a plus de littéraux à résoudre et plus de choix à traiter.

L'exploration de toutes les possibilités, backtrack et stratégie standard (L.Gacôgne, 2001)
Il y a retour en arrière (remontée dans l'arbre) chaque fois que toutes les règles ont été examinées sans unification possible, ou bien on arrive à une feuille de l'arborescence donnant un résultat, ou encore lorsqu'une impossibilité est bien notifiée dans la base de règles pour forcer la remontée, c'est "l'impasse". Ainsi si chaque descente dans l'arbre est associée à une transformation du but et à une "instanciation" des variables, chaque recul correspond à l'annulation de cette transformation. En fait le but étant examiné de gauche à droite, une seule sortie aura lieu pour cet exemple.
La stratégie standard est l'ordre de parcours racine-gauche-droite de l'arbre de recherche, cette stratégie est incomplète car en cas de branche infinie, une branche délivrant un succès risque de ne pas être atteinte, ainsi si on demande l'appartenance de a à L où a est une constante et L, l'inconnue, il existe une infinité de solutions L.
L'ordre des clauses a également son importance en cas de définition récursive.

5.4. Bibliothèques

Jusqu'aujourd'hui, les différentes éditions de PROLOG ne cessent de développer de nouvelles bibliothèques à ce langage pour qu'il s'étende afin qu'il réponde de plus en plus aux exigences de la programmation logique avec toutes ses branches récentes.

En sachant qu'il y a encore des tant de bibliothèques PROLOG, nous allons présenter les bibliothèques dont nous auront besoins dans notre travail.

5.4-1. CLP(FD) *(Triska)*

CLP (FD) est une bibliothèque qui étend la programmation logique avec le raisonnement sur des domaines spécialisés (les domaines finis). Elle nous permet de raisonner sur les entiers.

Il existe deux principaux cas d'utilisation de cette bibliothèque :

- Les variables de CLP(FD) mettent en œuvre des relations pures entre les expressions entières et peuvent être utilisés dans toutes les directions, même si les parties d'expressions sont des variables.
- Dans le cadre de prédicats d'énumération et plus de contraintes complexes, CLP (FD) est souvent utilisé pour modéliser et résoudre des problèmes combinatoires telles que les tâches de planification, d'ordonnancement et de répartition.

Pour pouvoir utiliser cette bibliothèque, il faut mettre l'instruction suivante dans le programme prolog : $:-use\_module(library(clpfd))$.
Il est à noter qu'il existe d'autres bibliothèques appartenant à la bibliothèque mère (CLP). En SWI-PROLOG, nous trouvons : CLP(B) (utilisées pour la programmation logique sur les variables booléenne), CLP(QR) (pour les variables rationnelles et les variables réelles).

5.4-2. SGML *(Wielemaker)*

Cette bibliothèque permet de transformer les données XML et HTML en une structure de données Prolog. Elle définit plusieurs familles de prédicats :

a. Prédicats de haut niveau : On utilise les prédicats load_html / 3, load_xml / 3 ou load_sgml / 3 pour transformer l'entrée et stocker le résultat dans une structure de DOM(Document Object Model)[1]. Ces prédicats tous les appels load_structure / 3, qui fournit plus d'options et peut être utilisé pour le traitement de documents non standard.
b. L'analyseur de bas niveau : Cet analyseur est écrit en C. Il se compose de deux parties : l'une offre le traitement DTD (Document Type Définitions) et l'autre sert à analyser les données. Les données peuvent soit être transformées en un terme Prolog (DOM) ou que l'analyseur peut effectuer des rappels pour les événements DOM.
c. Prédicats utilitaires : Ce sont des primitives qui servent à classer les caractères et les chaînes de caractères en fonction de la spécification XML tels que xml_name / 1 pour vérifier si un atome est un nom XML valide (identifiant). Ils fournissent également des primitives pour citer les attributs et éléments CDATA (Character Data)[2].

Pour pouvoir utiliser cette bibliothèque, il faut mettre ajouter cette instruction : $:-use\_module(library(sgml))$.

5.4-3. XPATH *(SWI-Prolog manual, library(xpath))*

La bibliothèque xpath.pl fournit des prédicats pour sélectionner des nœuds à partir d'un arbre DOM XML en tant que produit par la bibliothèque (sgml) basé sur des descriptions inspirées par le langage XPATH.

Le prédicat XPath / 3 sélectionne une sous-structure du DOM non-déterministe basé sur une spécification XPath-like.

Pour pouvoir utiliser cette bibliothèque, il faut mettre ajouter cette instruction : $:-use\_module(library(xpath))$.

6. Conclusion

D'après ce que nous avons vu au-dessus, la programmation logique par contraintes offre une résolution efficace aux problèmes de satisfaction de contraintes, ce qui réduit l'effort de concevoir à chaque problème à résoudre son propre algorithme et le coder après. Elle représente aussi un moyen très efficace à la résolution des problèmes combinatoires et notamment les problèmes d'ordonnancement (comme le problème de génération d'emploi du temps) en fournissant de multiples stratégies de résolution.

Le langage PROLOG est adapté à la programmation logique par contraintes. Les versions SWI-PROLOG sont libres. Elles sont devenues de plus en plus interopérables avec les langages de programmation classiques tels que JAVA, C++, …

PROLOG ne cesse pas de se progresser avec le développement de la programmation logique par contraintes. C'est pour ces raisons que nous avons choisi PROLOG pour se charger de la résolution du problème de génération d'emploi du temps universitaire.

[1] Le DOM XML présente le fichier XML sous une structure d'arborescence et permet au programmes d'accéder au document XML et de le modifier.

[2] Cette notation fait partie de la syntaxe XML. Elle permet au programme qui va analyser le fichier XML de seulement récupérer et non pas analyser ni interpréter ce qu'il y a entre le <![CDATA[et le]]>

CHAPITRE 03 : SPECIFICATION DES BESOINS ET CONCEPTION

1. Introduction

Afin de bien réaliser notre future application, il nous d'abord faut préciser les fonctionnalités qu'elle doit fournir. C'est pour cela que nous allons fixer tous les besoins des différentes parties prenantes (futurs utilisateurs). Ensuite, il nous faut une conception générale et détaillée de tout le système. C'est une phase primordiale dans tous les systèmes de production y compris le génie logiciel. Réussir à avoir une bonne conception rend plus simple la phase du codage ainsi celle de maintenance.

Dans ce chapitre, nous allons mentionner tous les besoins que nous devrons prendre en considération afin de les satisfaire tous les utilisateurs de l'application. Nous allons présenter par la suite une conception générale du système et finirons par une conception détaillée.

2. Spécification des besoins

A partir des exemples de résolution du problème d'emploi du temps cités dans le chapitre 1, et après avoir extrait les informations nécessaires au niveau du pédagogique, nous venons d'éliciter tous les besoins ainsi toutes contraintes que nous avons dues prendre en charge avant la conception de notre système.

Avant tout, on doit spécifier ce que doit offrir notre futur produit. Le système doit être capable de générer pour chaque département un emploi du temps valide qui repend aux besoins des enseignants et des étudiants tout en satisfaisant les exigences liées aux locaux qui lui appartiennent.

En analysant le cas général de tous les départements de l'université, on trouve que chacun d'eux doit occuper un certain nombre de locaux de différents types pour réaliser les différents types des séances hebdomadaires. De ce fait, il nous a fallu préciser les types de chaque entité.

Pour les séances, il existe trois types : les séances des cours, des travaux dirigés et des travaux pratiques. Chaque type se dérouler dans un type spécifique de locaux.

Les locaux se divisent sous quatre catégories : les amphithéâtres, les salles des travaux dirigés, les laboratoires des travaux pratiques et les salles d'informatique.

Chaque type de séance doit correspondre au type du local. Les cours doivent (en général) être enseignés au amphithéâtres.

Pour faciliter la spécification des contraintes, nous les avons classifiés sous trois catégories :

- **Contraintes de enseignants :**
  - Un enseignant ne peut enseigner qu'une séance à la fois.
  - La charge de l'enseignant (journalière et hebdomadaire) ne doit pas être dépassée.
  - Un enseignant peut enseigner plus qu'une spécialité et pourrait enseigner dans plus qu'un département. On doit prendre alors en considération les journées d'occupation des enseignants pour chaque département.
- **Contraintes des étudiants :**
  - Un étudiant ne peut figurer que dans un seul groupe.
  - Un groupe contient un nombre limité d'étudiants.
  - Un groupe ne peut avoir qu'une seule séance dans un créneau horaire précis.

- Un groupe pourrait appartenir à une section lorsque le département organise un certain nombre de groupes sous une section. Dans ce cas, un groupe ne peut appartenir qu'à une seule section.
- Dans ce dernier cas, on doit respecter l'homogénéité entre les sections et les groupes ; quand une section est concernée par un cours, alors tous les groupes qui la constituent doivent l'être et aucun d'eux ne peut se charger d'une autre séance.
- Les groupes et les sections appartiennent à des promotions.

- **Contraintes des locaux :**
  - Un local ne doit pas contenir un nombre d'étudiants supérieur à sa capacité.
  - Les cours en général doivent avoir lieu dans des amphithéâtres.
  - Un local peut être occupé par au plus une entité à la fois : une salle ne peut être occupée que par un seul groupe, un amphithéâtre ne peut être occupé que par une seule section, …
  - Un bloc contient des laboratoires de travaux pratiques, des salles de travaux dirigés et des salles d'informatique.

3. Conception

Dans le processus de conception du futur système, nous nous sommes servi du langage standard de modélisation UML (Unified Modeling Language) ; ce qui offre une vue multidimensionnelle précise sur les différentes parties composant les systèmes.

3.1. Diagramme de cas d'utilisation

Tout d'abord, nous donnons une brève description sur l'application et les différentes fonctionnalités qu'elle peut fournir. Pour cela, nous présentons le diagramme de cas d'utilisation (use case) qui se charge de cette activité.

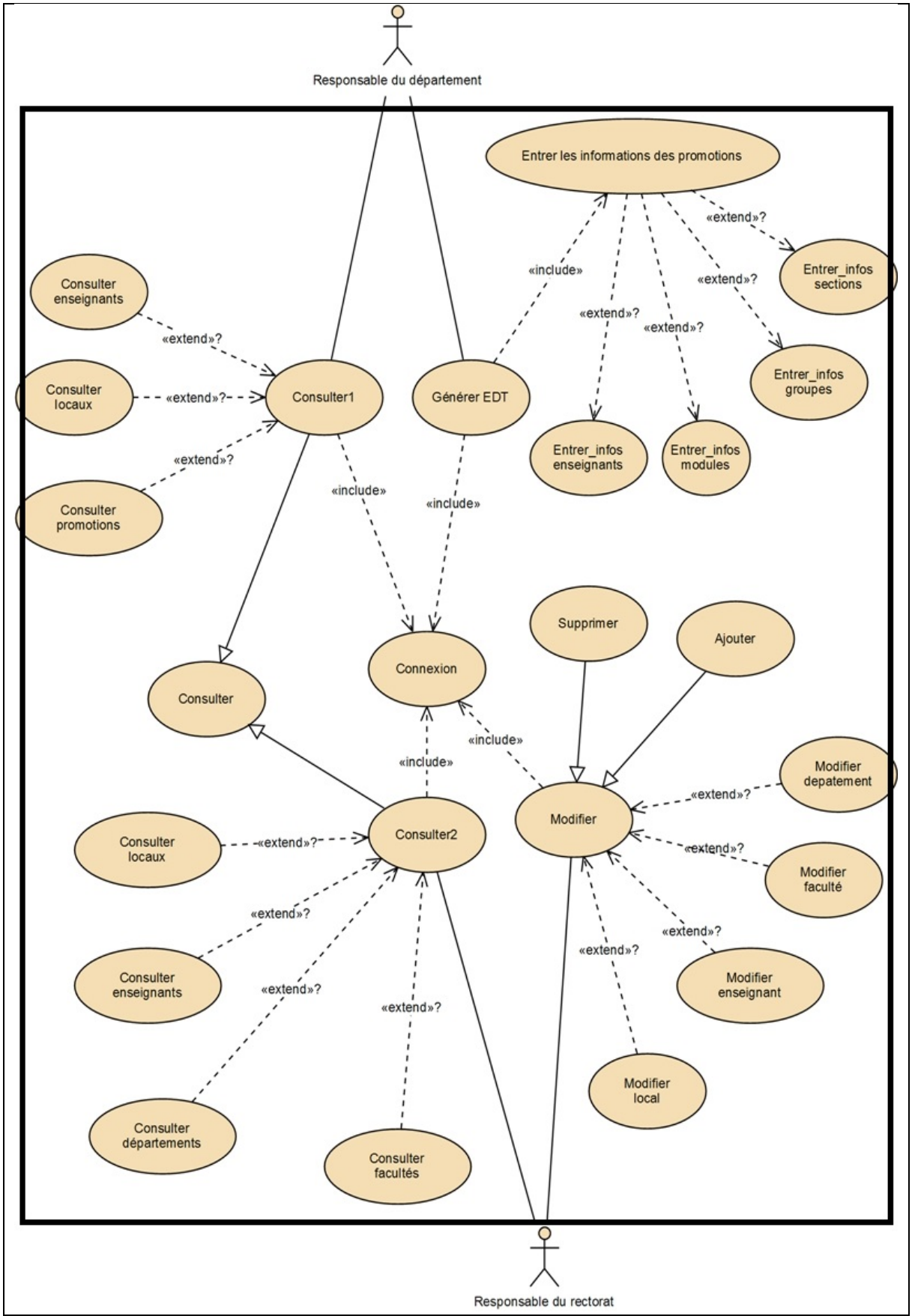

fig 6 Diagramme de cas d'utilisation

A partir de ce diagramme, on peut résumer toutes les fonctionnalités fournies par les différentes tierces faisant partie de l'application.

Premièrement, nous représentons les acteurs, ce sont les utilisateurs, ceux qui font marcher le système, autrement dit ceux qui le déclenchent et entrent les données requises afin que ce système puisse générer l'emploi du temps. En détaillant les rôles des acteurs dans notre cas, on trouve qu'il y'en a deux types :

a. Le responsable pédagogique du rectorat : c'est celui qui se charge de :
- consulter gérer (ajouter, modifier, supprimer) les facultés, départements, locaux et enseignants ;
- affecter les blocs au départements.

b. Les responsables pédagogiques des départements : il se charge de :
- consulter et gérer les promotions (ajouter, modifier, supprimer) ;
- consulter et gérer les locaux (affecter les locaux aux promotions qui appartiennent au département) ;
- consulter et gérer les enseignants appartenant au département (leur affecter les modules) ;
- générer l'emploi du temps du département.

Maintenant, nous identifions les activités pouvant être faites par les différents utilisateurs. En détaillant un peu leurs natures, on trouve que toutes les activités sont classées sous trois catégories : la consultation, la gestion et la génération.

(1) Consultation : c'est la fonction de superviser les données entrées par les différents utilisateurs ainsi que les emplois du temps générés.

(2) Gestion : c'est l'ensemble des fonctions à partir lesquelles les utilisateurs peuvent changer des données, ce qui aura forcément l'impact de changer des futurs emplois du temps.
La gestion comporte elle-aussi trois types : l'ajout, la suppression et la modification des données.

(3) Génération : c'est le processus de créer un calendrier au niveau de chaque département. Cela est fait après avoir rempli toutes les informations requises.

3.2. Diagramme de classes

Après l'analyse des cas d'utilisation, nous avons réalisé qu'il y avait deux types d'objets qui entrent en jeux dans le système : des objets actifs et des objets passifs.

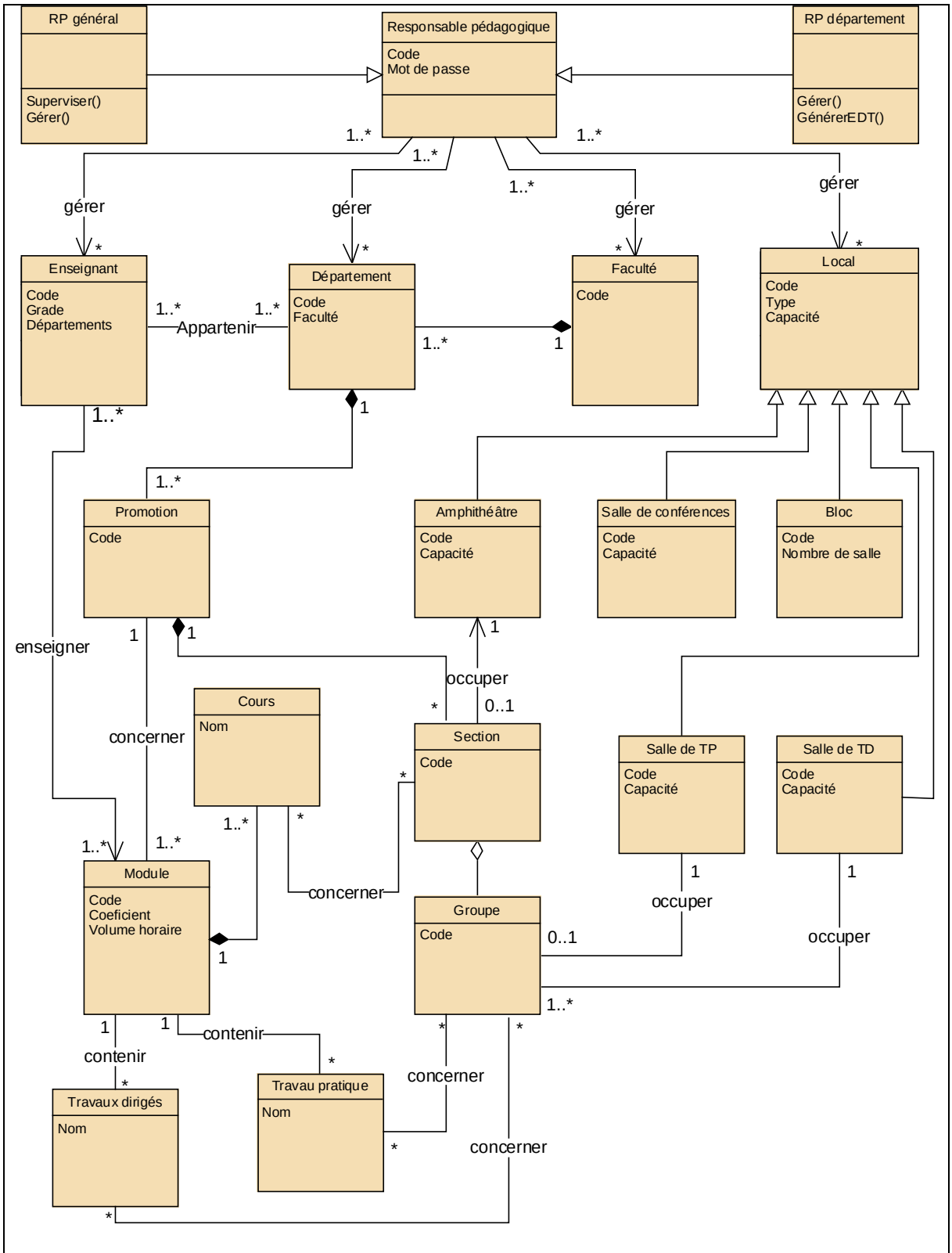

Fig 7 Diagramme de classes

Les objets actifs sont les objets qui agissent sur l'état du système soit par la consultation ou la modification. Tandis que les objets passifs, ils ne représentent que des données à utiliser.

Dans notre cas, les objets actifs sont les utilisateurs de l'application (responsable général et responsables du rectorat). Toutes les autres classes représentent les objets passifs.

3.3. Diagrammes d'activités

Les diagrammes d'activités décrivent les actions des différents objets et les flux des processus.

A partir du diagramme de cas d'utilisation, nous avons pu définir pour chaque acteur (responsable général et responsables des départements) les activités qu'ils peuvent exercer. Comme on a déjà vu, toutes les activités sont gradées au sein des trois catégories (consultation, modification et suppression), nous allons définir pour chaque type un seul exemple de diagramme d'activités qui représente un « modèle » sur lequel toutes seront basées les autres activités similaires.

3.3-1. Ajouter un enseignant

Afin que le responsable général puisse ajouter un enseignant, il doit d'abord entrer dans son espace dans l'application

- Se connecter ;
- Aller à la liste des enseignants ;
- Choisir l'action « Ajouter » ;
- Remplir le formulaire des informations de l'enseignant ;
- Valider le formulaire ;
- Dans le cas normal, l'enseignant sera ajouté.

Le diagramme ci-dessous modélise la description textuelle du processus d'ajout d'un enseignant. De la même façon nous avons conçu les autres diagrammes d'activités.

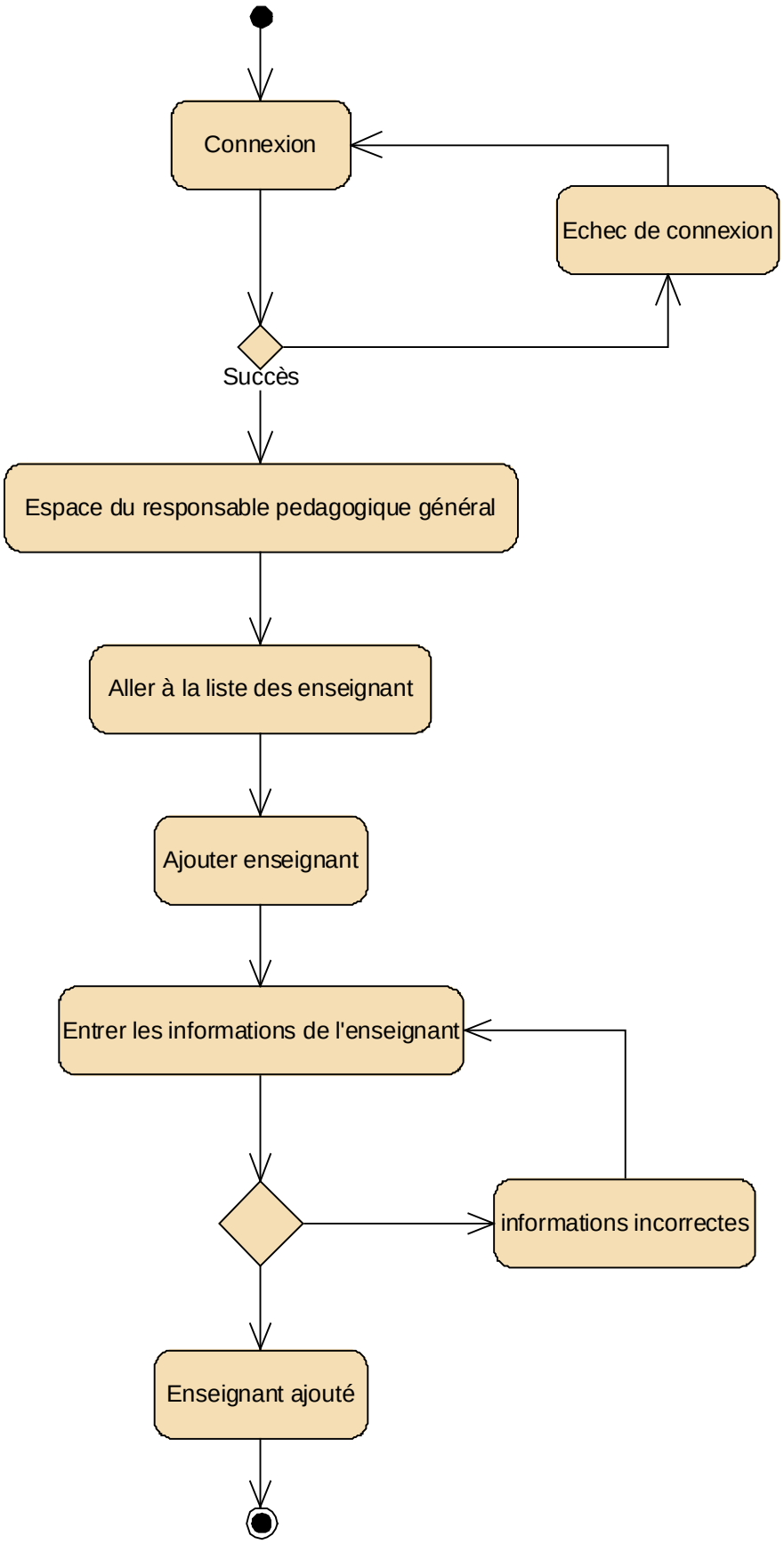

fig 8 Diagramme d'activités : Ajouter un enseignant

3.3-2. Supprimer un enseignant

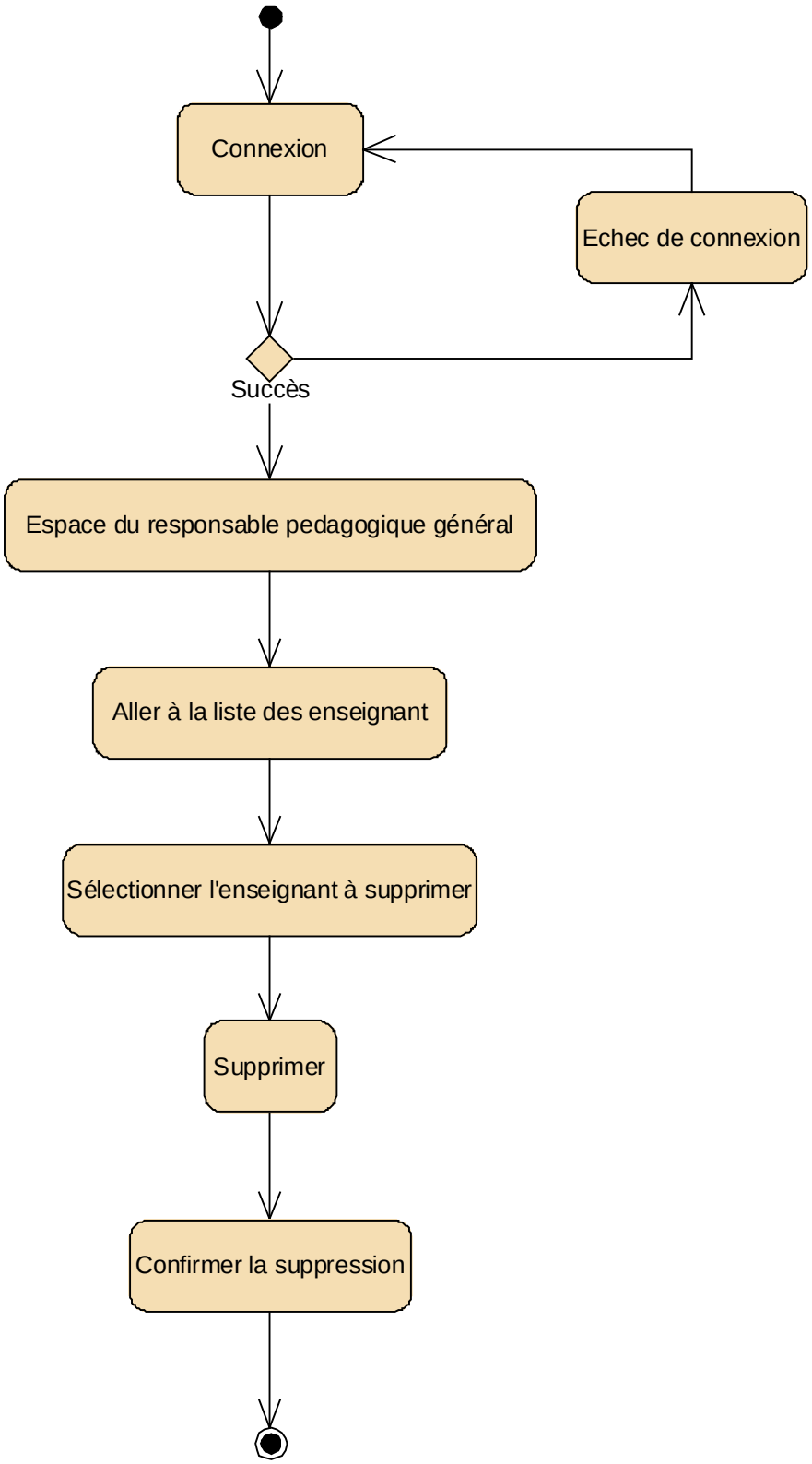

Fig 9 Diagramme d'activité : Supprimer un enseignant

3.3-3. Modifier un enseignant

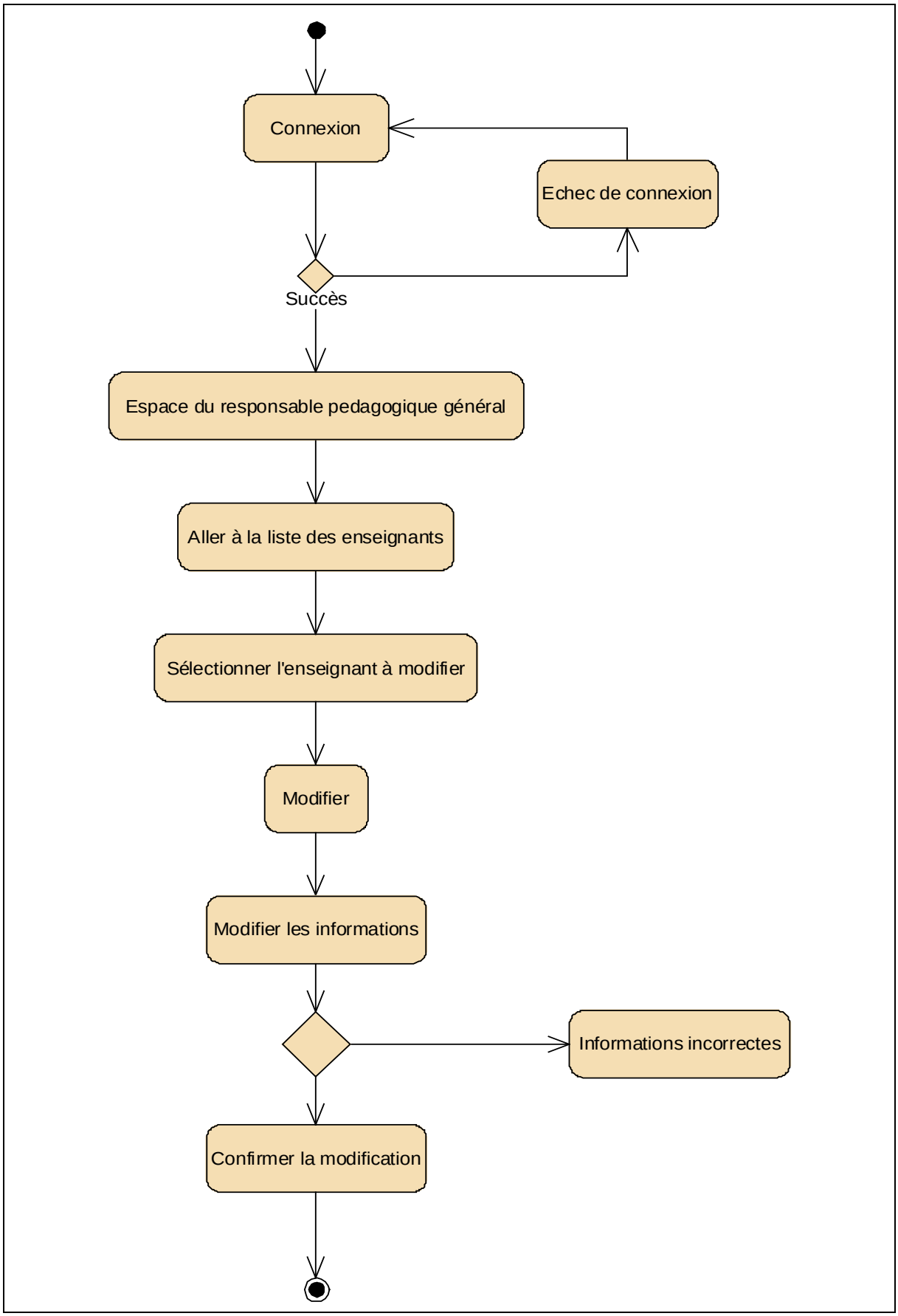

fig 10 Digramme d'activités : Modifier un enseignant

3.3-4. Génération d'emploi du temps

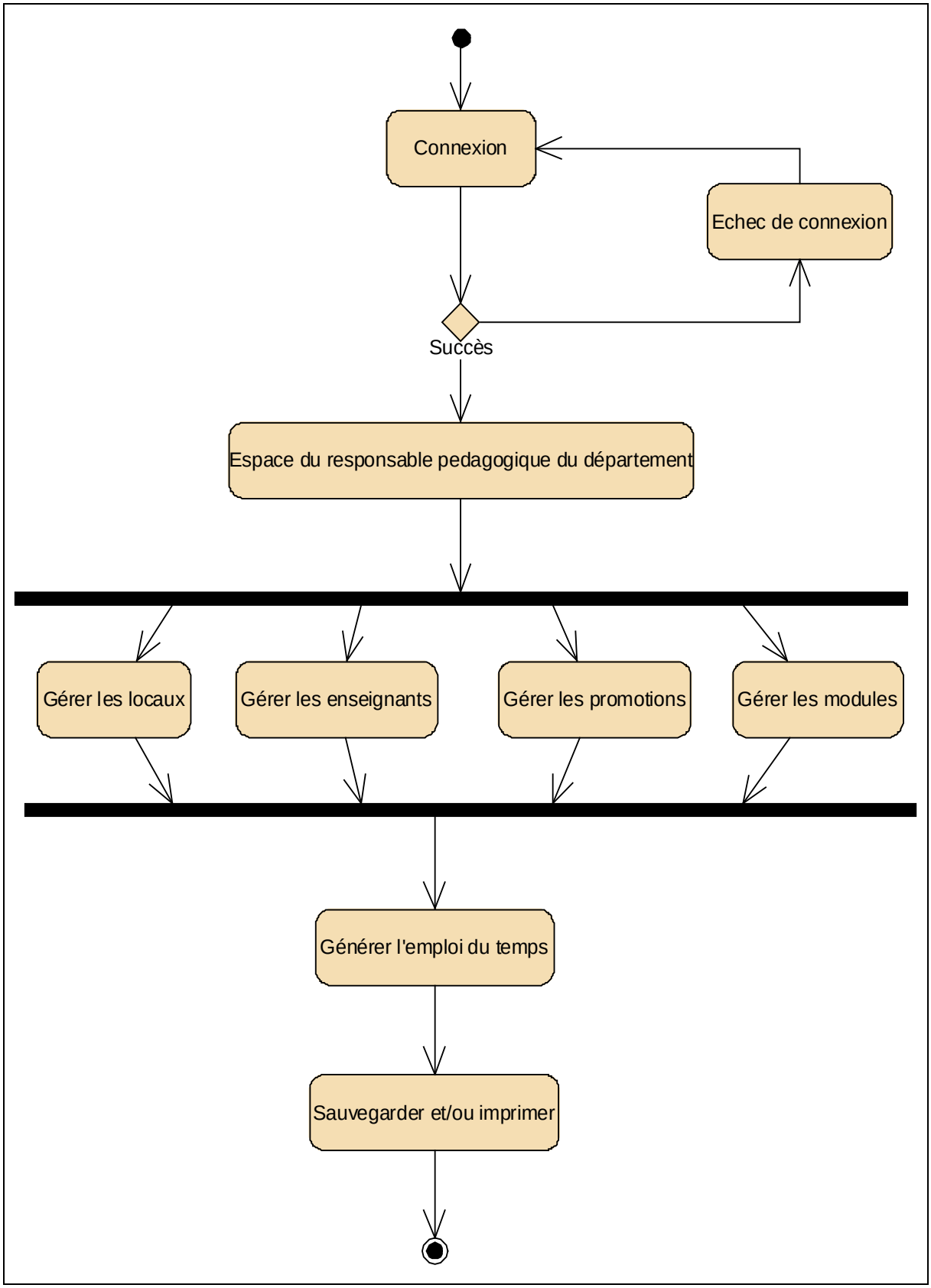

Fig 11 Diagramme d'activité : Générer l'emploi du temps

Il est à noter que l'activité de suppression est transitive, par exemple :

- La suppression d'une faculté implique la suppression de tous les départements qui l'appartiennent.
- La suppression d'un département implique la modification des informations des enseignants qui l'appartiennent.

3.4. Diagrammes de séquence

Les diagrammes de séquences décrivent les chemins d'exécution des processus en séquence ; c'est-à-d. l'ordre des opérations et les interactions entre les différents objets. On peut dire que ce type des diagrammes sert à raffiner toutes les fonctionnalités du système afin de donner une vue un peu proche à l'étape du codage (méthodes).

3.4-1. Connexion

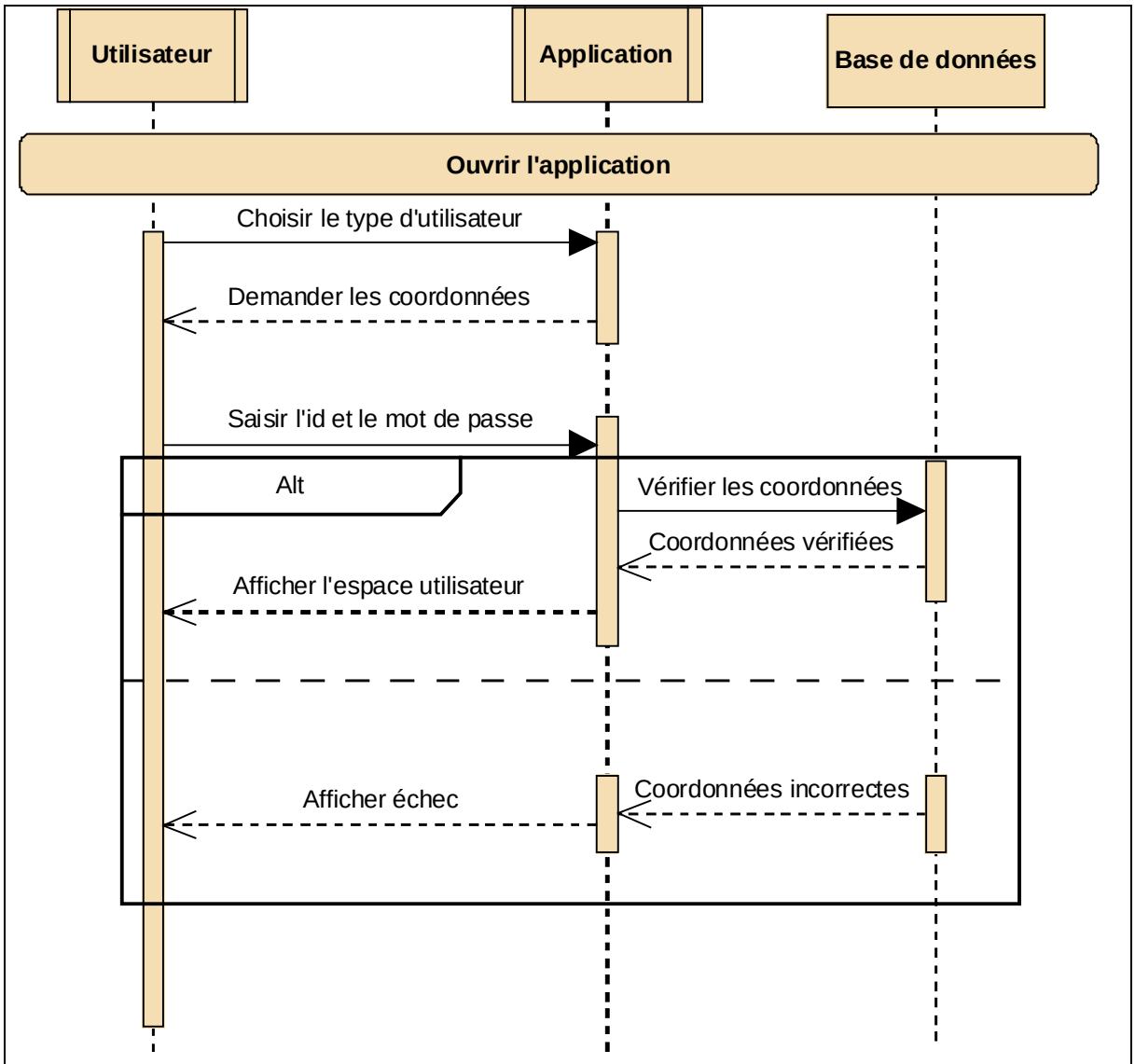

fig 12 Diagramme de séquences : Connexion

3.4-2. Modifier un compte

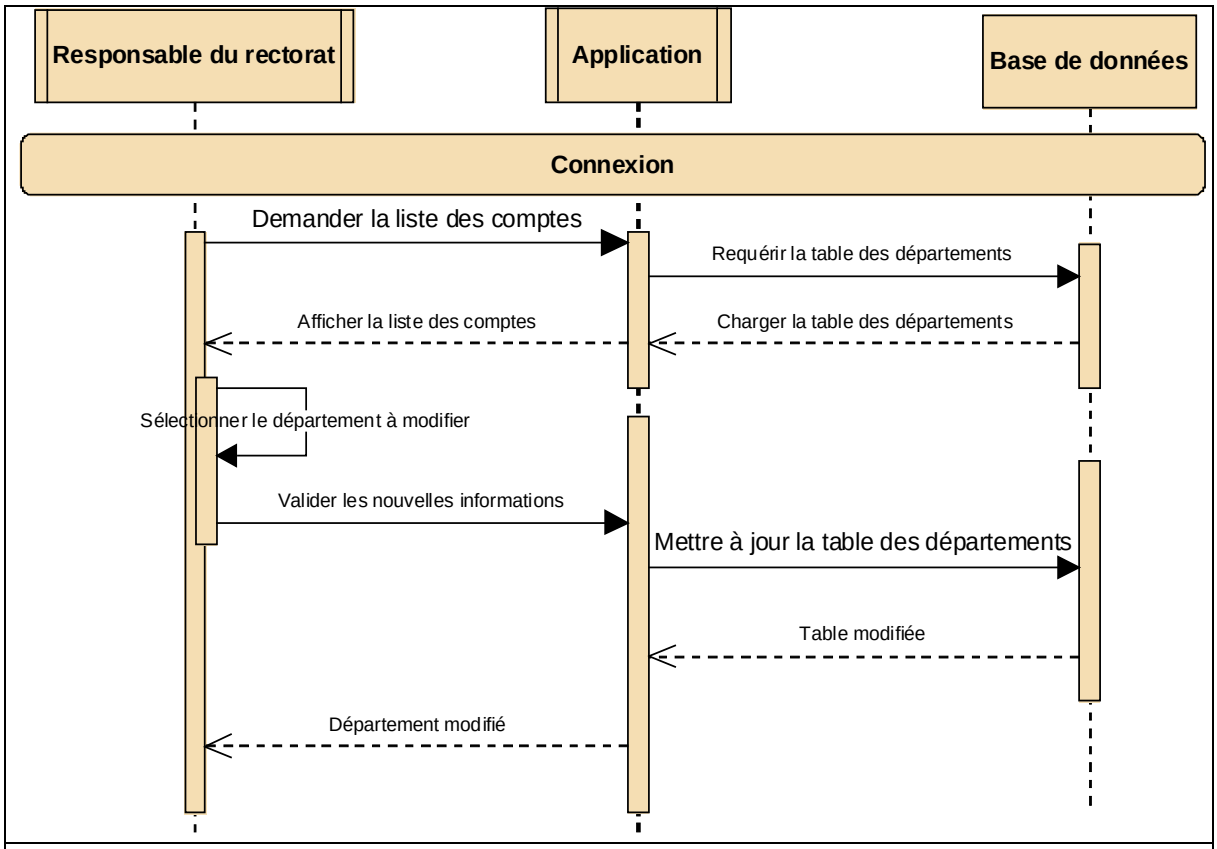

Fig 13 Diagramme de séquences : Modifier un compte

4. Conclusion

La spécification des besoins nous a aidés à bien concevoir notre future application. Cette conception va nous simplifier à nous mettre en place au moment du codage. Cela réduit l'effort de programmation en choisissant le paradigme adéquat et en divisant l'application en modules.

Après que nous ayons conçu notre système, nous allons mettre en œuvre l'application. Le chapitre suivant traitera ce fait.

CHAPITRE 04 : REALISATION ET EXPERIMENTATIONS

1. Introduction

Après que nous ayons conçu notre système, nous avons fini par choisir la méthode que nous devions suivre dans la phase de codage afin de bien réaliser notre application ainsi les outils adéquats à cette tâche.

Dans ce chapitre, nous allons présenter les outils que nous avons utilisés lors de cette étape. Ces outils comprennent les langages de programmation et les environnements de développement utilisés. Nous décrierons le processus de fonctionnement de l'application réalisée. Et nous finirons par une expérimentation.

2. Outils utilisés

Pour réaliser notre application, nous avons utilisé trois langages de programmation de deux différents paradigmes : JAVA, SQL et PROLOG avec une interopérabilité entre eux. JAVA a été choisi pour se charger de l'interaction homme machine, c.-à-d. qu'il offre une interface graphique qui facilite la communication entre l'utilisateur et l'application. Le deuxième langage (SQL) traite les requêtes venues de java pour les opérations sur la base de données. PROLOG s'occupe de la résolution après la récupération de toutes les données requises.

2.1. Langages de programmation et environnements

a. JAVA

JAVA nous offre une base puissante pour faciliter le développement des interfaces. Ces dernières comprennent :

- ✓ les interfaces homme/machine qui servent à communiquer l'application avec les utilisateurs de façon simple et de sorte qu'il n'apparaisse à eux que les fonctionnalités dont ils ont besoins, c.-à-d. que les utilisateurs ne s'inquiètent pas comment fonctionne l'application au fond ;
- ✓ la modularité des composants ; chaque composant est responsable d'une tâche. Ça nous a permis de traiter rapidement les erreurs. La modularité a été réalisée grâce au paradigme orienté objet offert par JAVA ;
- ✓ la communication à la base de données. ;
- ✓ la communication avec PROLOG.

b. SQL

Nous avons utilisé une base de données pour stocker les informations nécessaires afin d'en offrir une réutilisabilité et d'éviter de remplir les informations à chaque fois qu'on veut générer un emploi du temps.

Nous avons utilisé le SGBD MySQL fourni par le serveur UWamp.

Notre base de données contient les tables suivantes :

- utilisateur : contient les informations (identifiants et mots de passe des utilisateurs de cette application (responsable pédagogique du rectorat et responsables pédagogiques des départements).
- liste_facultes : contient les informations de toutes les facultés.

- liste_departements : contient les informations des départements et les facultés auxquelles ils appartiennent.
- liste_locaux : contient la liste des différents types des locaux.
- liste_enseignants : contient la liste des enseignants et les départements auxquels ils appartiennent.

c. PROLOG

Après la collection des données nécessaires pour la génération d'emploi du temps au niveau de chaque département, PROLOG est utilisé pour la résolution. Il offre pour cette étape plusieurs stratégies. La stratégie fournie par défaut est celle du retour en arrière (backtrack). On peut l'utiliser en ajoutant l'instruction : $labeling([leftmost], [Vars])$ ou tout simplement : $labeling([\,], [Vars])$.
$Vars$ est l'ensemble des variables sur lesquelles on veut appliquer cette stratégie.

Nous allons présentons les différentes stratégies qui existent dans les SWI-PROLOG :

$leftmost$	Etiqueter les variables dans l'ordre tel qu'elles sont apparues à Vars. Ceci est le cas par défaut.
ff	(First fail) : Etiqueter d'abord la variable de gauche avec le plus petit domaine, afin de détecter tôt l'infaisabilité si ce sera le cas.
ffc	Parmi les variables avec les plus petits domaines, la plus à gauche qui participe à la plupart des contraintes est marquée d'abord.
min	Etiqueter d'abord la variable à gauche dont la borne inférieure est la plus petite.
max	Marquez d'abord la variable à gauche dont la borne supérieure est la plus élevée.

Prenons l'exemple suivant :

$:- use\_module(library(clpfd))$.

$resoudre(Vars) :-$

$Vars = [X, Y]$,

$Vars\ ins\ 1..3$,

$labeling([ff], [X, Y])$.

Si on exécute la requête suivante : $?- resoudre([X, Y])$, le résultat est :

$X = 1, Y = 1$

Le solveur choisit pour chaque variable la première valeur de son domaine.

Si, par contre, on change la stratégie, on aura des résultats différents. Par exemple :

$labeling([min(X), max(Y)], [X, Y])$.

Le résultat est :

$X = 1, Y = 3$

Dans notre problème, toutes les stratégies offrent un résultat. Le choix de la meilleure stratégie dépend du temps écoulé pour la résolution. Et comme la génération ne prend qu'une seconde, nous ne pouvons découvrir manuellement qu'une stratégie est meilleure qu'une autre.

2.2. Environnement de développement

Pour développer en JAVA, nous avons utilisé l'environnement ECLIPSE. L'utilisation des plug-ins est très facile dans cet environnement.

Pour les plug-ins, nous avons utilisés :

- MySQL connector : pour établir la connexion entre l'application et la base de données.
- RS2XML : Lorsqu'on veut stocker l'ensemble des résultats d'une requête SQL dans une table en JAVA (JTable),
- COMMONS-IO : La sortie de génération de PROLOG est un fichier txt. Pour pouvoir le stocker dans une JTable, il faut d'abord le transformer en chaine de caractères. Cette tâche est effectuée par ce plug-in.

3. Description de l'application

Notre application contient deux espaces : espace resposable pédagogique du rectorat et espace responsable pédagogique du département.

Pour qu'un utilisateur puisse accéder à son espace, il doit d'abord se connecter via ses coordonnées (nom d'utilisateur et mot de passe). La figure ci-dessous montre l'accueil de notre application permettant la connexion.

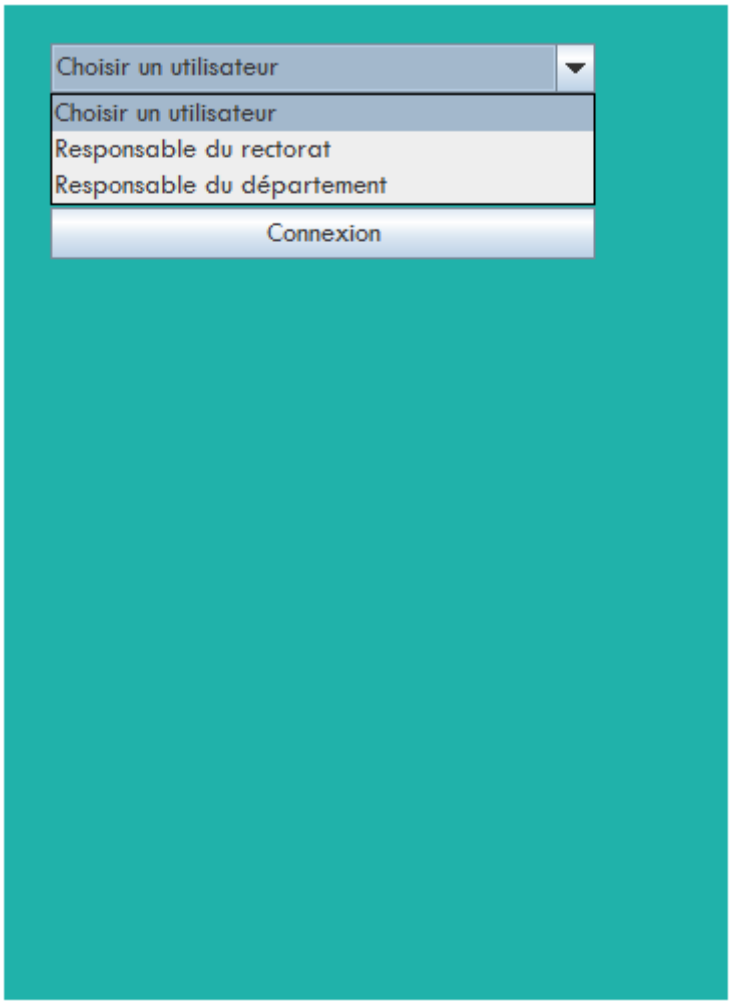

fig 14 Accueil de l'application

Après la connexion, chaque utilisateur est dirigé vers sont espace. Prenons l'exemple de l'espace du responsable pédagogique général (du rectorat).

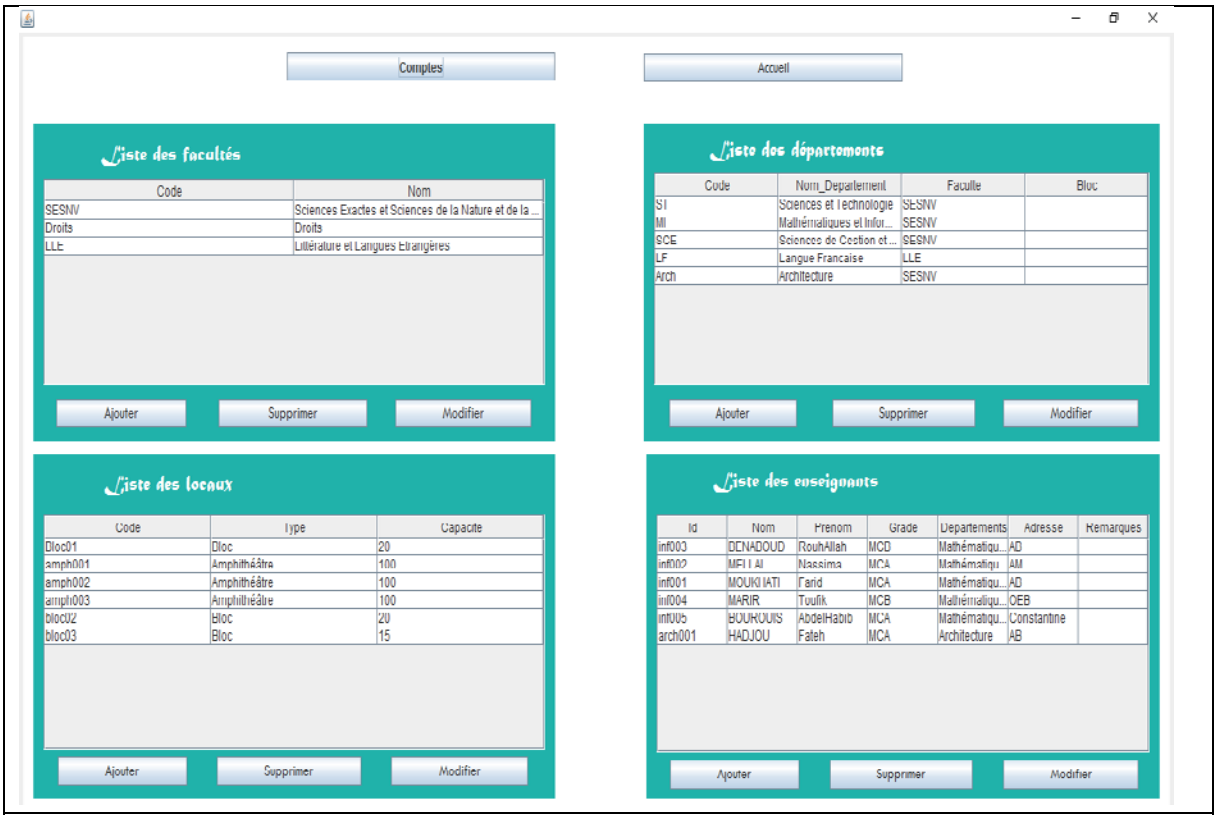

fig 15 Espace du responsable pédagogique du rectorat

On remarque que cet espace contient les listes des informations générales des facultés, des départements, des locaux et des enseignants. Le responsable pédagogique général est capable de consulter, de modifier et de supprimer ces informations.

Cet espace contient en outre la liste des comptes qui comporte les coordonnées de tous les utilisateurs de l'application. Le responsable général doit toujours ajouter un compte lors d'une opération d'ajout d'un département. Ce compte permet au responsable du département d'accéder à son espace.

La suppression d'un département inclut la suppression du compte de responsable de ce département.

Génération d'emploi du temps : La tâche génération d'emploi du temps est effectuée par le responsable du département.

Le processus de génération est décrit dans la figure suivante :

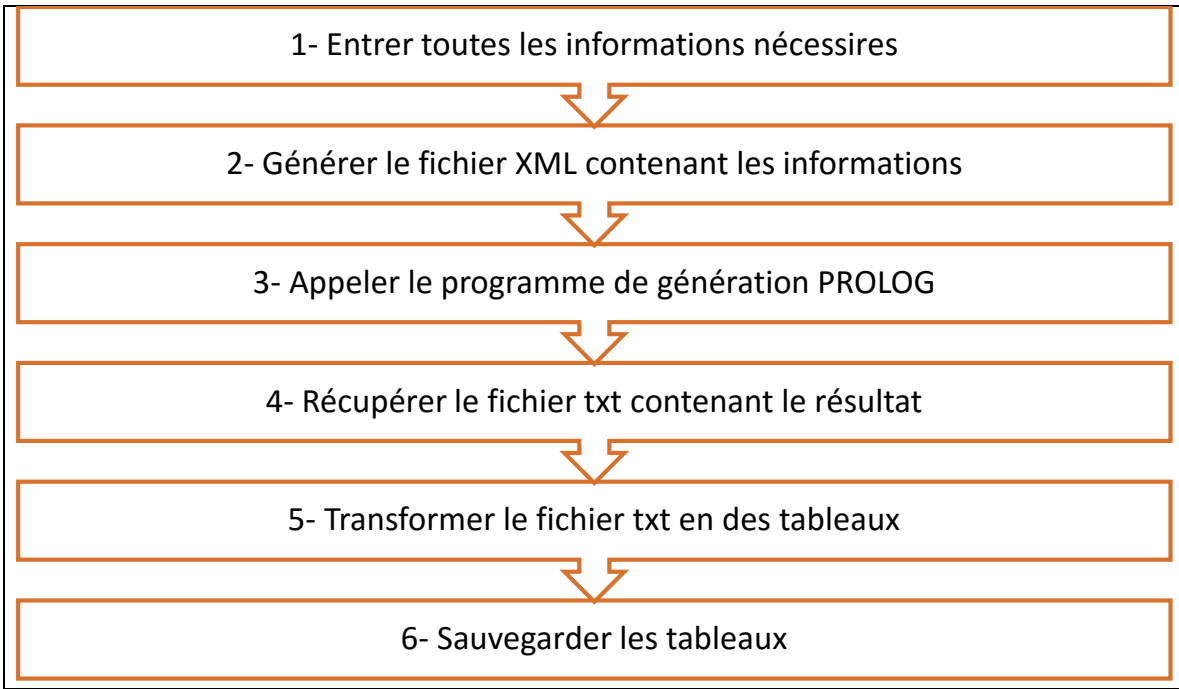

Fig 16 Processus de génération d'emploi du temps

La structure du fichier XML contenant les informations nécessaires à la génération est comme indiquée dans la figure ci-dessous.

```
<?xml version="1.0" encoding="UTF-8"?>
<besoins>
    <global numCHs="36" CHParJour="6"/>
    <groupe id="g1">
        <module cid="module1" enseignant="enseignant1" creneauHoraire="4"/>
        <module cid="module2" enseignant="enseignant2" creneauHoraire="7"/>
        ....
    </groupe>
    <groupe id="gn">
        <module cid="module1" enseignant="enseignant1" creneauHoraire="4"/>
        <module cid="module2" enseignant="enseignant2" creneauHoraire="7"/>
        ....
    </groupe>
    ....

    <local id="l1">
        <allouer groupe="g1" cid="cours1" seance="0"/>
        ....
    </local>
    <journeeLibre enseignant="enseignant1" jour="3"/>
    ....

</besoins>
```

fig 17 Structure du fichier XML des informations

4. Expérimentation

Le résultat est stocké dans des tableaux. Il peut être vu selon les enseignants et selon les groupes. Des exemples de résultats que nous avons capturés sont illustrés dans les figures suivantes (Ceci est un exemple où nous n'avons pas respecté le volume horaire des enseignants et des groupes).

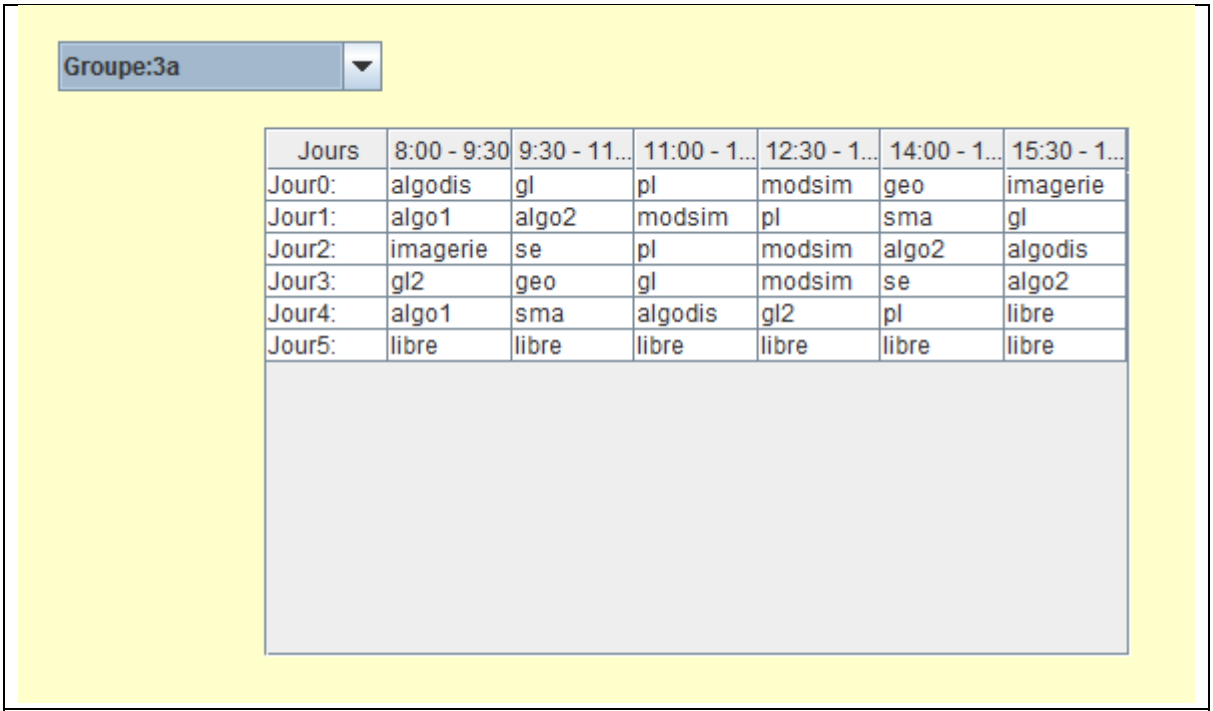

Fig 18 Exemple d'un emploi du temps d'un groupe

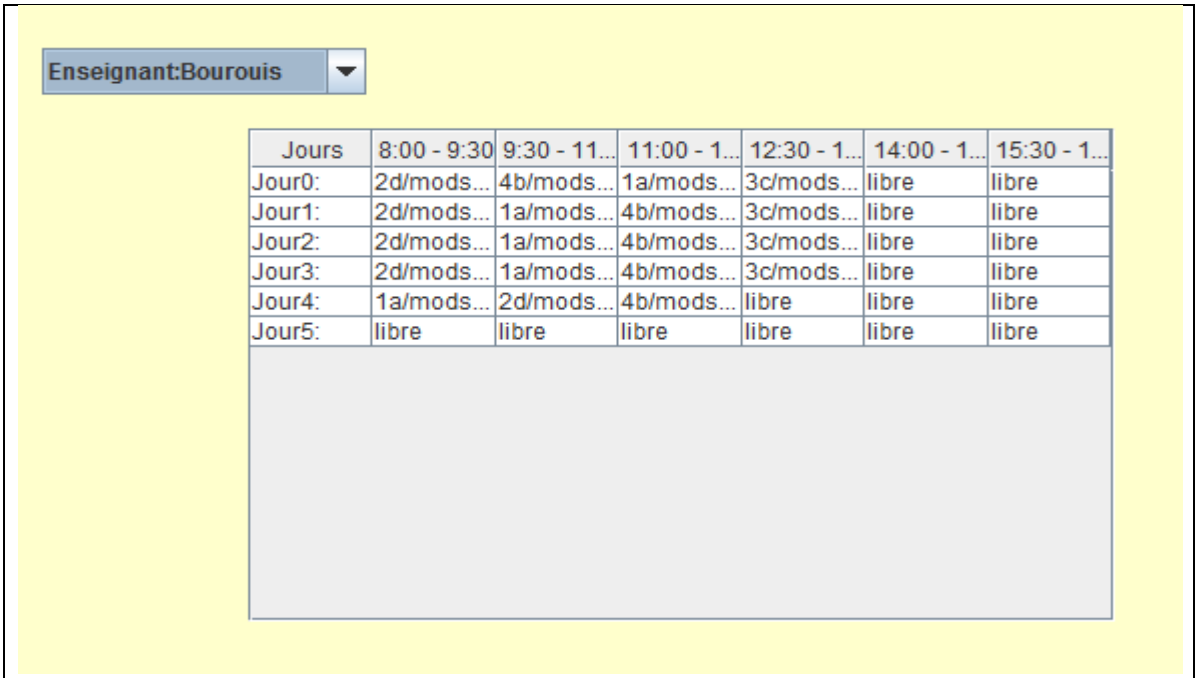

fig 19 Exemple d'un emploi du temps d'un enseignant

5. Conclusion

Dans notre approche, nous avons séparé les données des traitements. La base de notre résolution dépend d'un fichier XML contenant les informations introduites par l'utilisateur et le fichier PROLOG contenant le programme de résolution (qui est fixe). De cette manière, la dynamité de l'application réside au niveau de données entrées par l'utilisateur mais pas au niveau de résolution. Cela permet d'éviter que l'application génère, à chaque fois qu'on veut générer un emploi du temps, le fichier PROLOG de résolution.

Pour garder la transparence du processus de génération d'une part, et faciliter la communication de l'application avec l'utilisateur d'une autre part, nous avons créé des interfaces graphiques à travers JAVA qui se charge lui-même, outre que créer ces interfaces, de transformer les données entrées par l'utilisateur en éléments d'un fichier XML.

Conclusion générale

Dans ce mémoire, nous avons proposé une approche basée sur les problèmes de satisfaction des contraintes pour résoudre le problème d'emploi du temps universitaire. Avec le progrès des outils de résolution logique pour ce type de problèmes, les développeurs ne s'inquiètent plus à élaborer des algorithmes pour les résoudre. Beaucoup de techniques ont été fondées pour se charger de ce travail.

Malgré la rapidité de résolution des problèmes d'emploi du temps universitaire en comptant sur la programmation logique par contraintes, il reste encore la possibilité d'optimiser le processus de construction de l'application dans sa totalité ou encore la capacité d'élaborer des applications distribuées en se basant sur cette approche.

Pour l'optimisation, plusieurs recherches ont été faites pour rendre facile l'interopérabilité JAVA PROLOG et minimiser l'effort du codage. Le projet CAPJA (Connector Architecture for Prolog and JAva) prouve notre perspective : c'est une architecture proposée (Ostermayer, 2015) qui consiste à intégrer les prédicats de PROLOG en JAVA et de transformer les termes en objets. Une expérimentation ayant été faite par l'outil JPLambda basé sur les expressions lambda JAVA (qui n'est pas encore officialisé) a réalisé que la moitié du code était réduite.

Pour la distribution, il faut penser à des conceptions réparties. Nous recommandons de proposer des architectures basées sur la répartition des données ou des traitements. Le premier cas serait relativement simple, le programme de résolution qui est écrit en PROLOG ne change pas, il faut juste fonder des algorithmes servant à réassembler les données réparties pour les faire passer par le programme de résolution. Il est recommandé, dans ce cas, de penser aux applications web, l'outil PDT (Prolog Development Tool) est développé pour faciliter l'interopérabilité de PROLOG avec différents langages de programmation (comme JAVA, PHP, … etc.). Dans le cas de distribution du traitement, la résolution deviendrait plus difficile et ne garantissait pas un résultat adéquat car outre que les données, c'est le programme de résolution qui change.

Bibliographie

Ben Rahou, T. (2013). *Nouvelles méthodes pour les problèmes d'ordonnancement cyclique.* Toulouse: Université Paul Sabatier.

Branimir Sigl, Marin Golub, Vedran Mornar. (s.d.). *Solving Timetable Scheduling Problem.* Zagreb, Croatia: Faculty of Electrical Engineering and Computing, University of Zagreb.

E. Aycan, T. Ayav. (s.d.). *Solving the Course Scheduling Problem Using.* Izmir: Département d'Ingénierie d'Ordinateur, Institut de Technologie.

Ferber, J. (1995).

HERITAGE, P. (2010). *Manuel de Prolog.*

Houria, B. (2012). *Un système automatique pour la génération des emplois du temps basé sur le logiciel national gestion de scolarité SEES.* Université de Tlemcen.

Houssem Eddine Nouri, Olfa Belkahla. (2014). *Résolution multi-agents du problème d'emploi du temps universitaire.*

J.Carlier, P.Chrétienne, J.Erschler, C.Hanen, P.Lopez, E.Pinson, M-C.Portmann, C.Prins. (1988). *LES PROBLEMES D'ORDONNANCEMENT.*

J-P.SANSONNET. (2005). Applications agents pour le commerce électronique. Université Paris Sud-XI.

L.Gacôgne. (2001). *PROGRAMMATION LOGIQUE, LE CHAINAGE-ARRIERE : PROLOG.*

LEMOUZY, S. (2011). *Systèmes interactifs auto-adaptatifs par systèmes multi-agents auto-organisateurs : application à la personnalisation de l'accès à l'information.* Ecole doctorale : Mathématiques informatique télécommunications de Toulouse.

MILI, R. (2010). *Conception et implémentation d'un algorithme d'optimisation pour la Génération automatique d'emploi du temps Universitaire.* Université de Skikda.

Olivia Rossi-Doria, Christian Blum,Joshua Knowles, Michael Sampels,. (s.d.). *A local search for the timetabling problem.* Edinburgh, Écosse.

Ömer S. Benli, A. Reha Botsali. (2004). Timetabling Problems. Dans *Detailed Design of a DSS for Course Scheduling in a University.*

Ostermayer, L. (2015). *Seamless Cooperation of Java and Prolog for Rule-Based Software Development.* University of Würzburg.

Pastre, D. (1999). *Prolog.* Université de Paris 5.

Patrick ESQUIROL, Pierre LOPEZ. (1995). *Programmation Logique avec Contraintes et Ordonnancement.* I.N.S.A. de Toulouse.

(s.d.). *SWI-Prolog manual, library(xpath).* www.swi-prolog.org.

Thierry Moyaux, Brahim Chaib-draa, Sophie D'Amours. (s.d.). *Satisfaction distribuée de contraintes et son application à la génération d'un emploi du temps d'employés.* Sainte-Foy, Québec: Centre de recherche sur les technologies de l'organisation réseau, Laboratoire DAMAS, Département d'Informatique et de Génie Logiciel, Faculté des Sciences et de Génie, Université Laval.

Triska, M. (s.d.). *SWI-Prolog manual, library(clp(fd)).* www.swi-prolog.org. Récupéré sur www.swi-prolog.org.

Wielemaker, J. (s.d.). *SWI-Prolog manual, library(sgml).* University of Amsterdam.

Wikipedia. (2015). *Clips (langage).*

Wikipedia. (2015). *Programmation logique.*

Wikipedia. (2016). Algorithme génétique.

Wikipedia. (2016). *Prolog.*

Wikipedia. (2016). Recherche locale (optimisation).

Wikipedia. (2016). Recuit simulé.

Wikipedia. (2016). Théorie de l'ordonnancement.

Wikipeida. (2016). Programmation par contraintes.