Allocation distribuée de ressources dans le système CONSENSUS

Sylvain Sécherre, Francois Fages

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

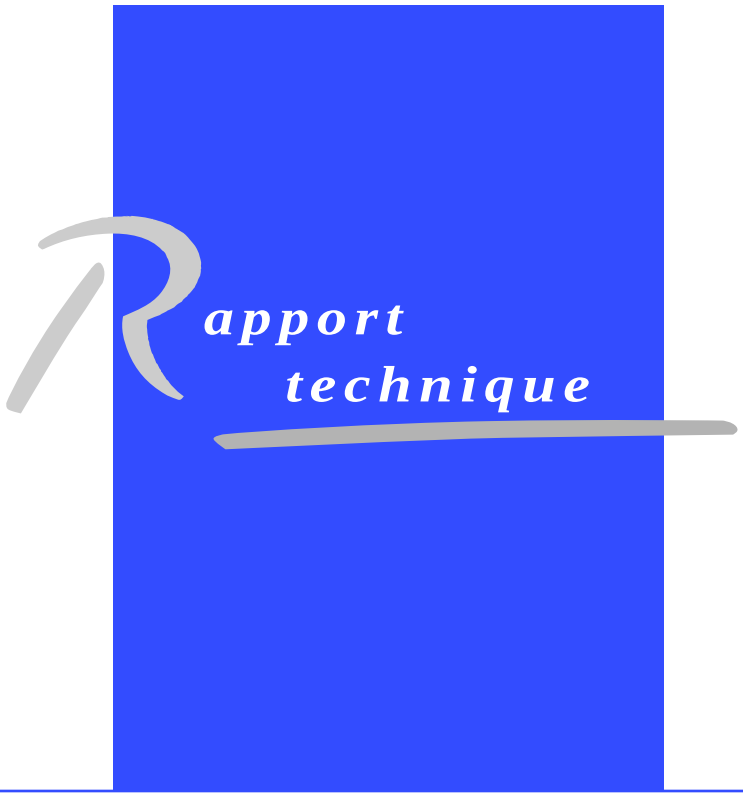
Allocation distribuée de ressources dans le système CONSENSUS

Sylvain Sécherre — François Fages

N° 0308

Juillet 2005

Thème SYM



ISSN 0249-0803

Allocation distribuée de ressources dans le système CONSENSUS

Sylvain Sécherre*, François Fages

Thème SYM — Systèmes symboliques
Projets Contraintes

Rapport technique n° 0308 — Juillet 2005 — 79 pages

Résumé : Ce rapport décrit la résolution d'un problème d'allocation de ressources sur-contraint avec le système multi-utilisateur CONSENSUS. Les salles d'un campus, réparties dans plusieurs bâtiments, sont à affecter à plusieurs équipes en fonction de leurs besoins. Les contraintes du problème sont les besoins des utilisateurs. Ces besoins ne seront pas forcément satisfaits, et expriment donc des préférences plutôt que des contraintes dures. De plus ces contraintes sont de différentes natures, et ne sont donc pas facilement comparables. Enfin, on ne fait pas l'hypothèse que les utilisateurs expriment toutes leurs préférences, celles-ci étant apprises à partir des notes qu'ils donnent aux solutions proposées par le système.

La méthode utilisée pour calculer des solutions est une méthode de recherche adaptative. Il s'agit d'une méthode de recherche locale Tabou permettant de résoudre des problèmes sur-contraints en recherchant une solution violant le moins possible l'ensemble des contraintes. Une difficulté particulière est de chercher à rendre les contraintes hétérogènes comparables afin que la méthode n'en favorise pas certaines plus que d'autres. L'apprentissage des préférences des utilisateurs est traité quant à lui par un programme linéaire.

La résolution globale du problème est itérative, faisant se succéder des phases de calculs de solutions (à partir des préférences exprimées et apprises), et des phases de notation des solutions par les utilisateurs, jusqu'à trouver une solution consensuelle, qui insatisfait le moins aucun utilisateur. D'un point de vue programmation, CONSENSUS est constitué d'un serveur Web et de clients légers, entièrement implanté en Html, XML, Perl et Scilab.

Mots-clés : optimisation combinatoire, allocation de ressources, recherche locale, préférences, contraintes, apprentissage

* Elève de l'ENSTA, stagiaire à l'INRIA de février à juillet 2005.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Distributed Allocation of Resources with the CONSENSUS system

Abstract: This report describes the solving of an over-constrained resource allocation problem using the multi-user system CONSENSUS. The rooms in the different builings of a campus must be assigned to teams, according to their needs. The constraints of the problem are the needs of the users. These needs are not necessarily satisfiable, and thus express preferences rather than hard constraints. Morover, these constraints are heterogeneous and thus not easily comparable. Finally, we do not assume that all users express all their preferences. They are learned from the notations given by the users to the solutions proposed by the system.

The method used to compute solutions is adaptive search, a Tabu local search method capable of solving over-constrained problems by searching for configurations violating a minimum set of constraints. One peculiar difficulty is to make heterogeneous constraints comparable, in such a way that the method does not focus on some constraints. The learning of the preferences of the user are dealt with a linear program.

The global resolution of the problem is iterative, alternating phases of computation of solutions (considering the expressed or learned preferences of the users), with phases of notations of the solutions by the users, until a consensus is found unsatisfying the least any user. From a programming point of view, CONSENSUS is composed of a Web server and light clients, it is entirely implemented in Html, Xml, Perl and Scilab.

Key-words: combinatorial optimization, resource allocation, local search, preferences, constraints, learning

1 Introduction

Le projet Contraintes est impliqué dans le projet `CONSENSUS` dont le but est la mise au point d'un outil qui facilitera la répartition des locaux entre les différentes équipes tout en satisfaisant au mieux leurs besoins. Cet outil doit permettre :

- aux utilisateurs d'exprimer leurs besoins, leurs préférences et de noter des solutions,
- de calculer des solutions et de les soumettre aux utilisateurs,
- d'estimer les préférences réelles des utilisateurs,

L'architecture de l'outil est à deux niveaux :

- Le premier s'appuie sur la méthode de recherche adaptative [2] et a pour but de fournir une solution au problème d'allocation compte tenu des préférences exprimées ou apprises.
- Le second niveau apprend les préférences de l'utilisateur en prenant en considération les notes attribuées aux solutions calculées.

Les deux niveaux s'enchaînent jusqu'à ce qu'une solution "consensuelle" soit trouvée, insatisfaisant le moins possible les utilisateurs.

Les besoins exprimés sont de plusieurs ordres : nombre de salles désirées, type de salles désirées, distance à un bâtiment, distance à une autre équipe, regroupement de l'équipe. Chaque besoin est assorti d'un coefficient de façon à permettre à une équipe de mettre l'accent sur tel ou tel besoin.

Ces besoins sont les contraintes du problème. On fait dès le départ l'hypothèse que ces contraintes pourront être violées. L'outil doit pouvoir donner une solution même si le nombre de salles demandées dépasse la capacité du campus.

Ce projet a déjà fait l'objet d'un stage et un développement en `C++` [3][6] en collaboration avec le projet Complex. La présente étude s'est concentrée sur la rédéfinition des fonctions d'erreur associées aux différentes contraintes de façon à s'assurer qu'elles sont évaluées de façon homogène, ce qui est essentiel pour ne pas biaiser (autrement que par les coefficients de préférences) la méthode d'optimisation locale en présence de contraintes hétérogènes. D'autre part, la partie apprentissage des préférences a été modélisée comme un problème d'optimisation linéaire, que l'on résoud de façon exacte par un programme linéaire. Pour ces raisons, le système CONSENSUS a été implanté sur des bases nouvelles en Scilab, Perl et Html. L'interface graphique s'appuie sur les technologies WEB clients 'légers', serveur 'lourd', de façon à faciliter l'utilisation de l'outil par plusieurs intervenants distants.

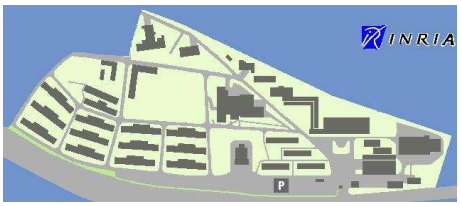


FIG. 1 – *Un système particulier : le campus de l'INRIA*

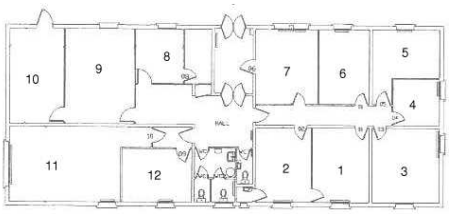


FIG. 2 – *Le bâtiment 35 du campus comprenant des salles de différents types.*

2 Problème d'allocation des ressources

2.1 Enoncé du problème

Le système désigne un ensemble de salles et un ensemble d'équipes.

Les salles (cf. Fig. 2)peuvent être de différents types suivant leur surface ou leurs équipements. Le type caractérise seul une salle et c'est à l'administrateur de réaliser la classification en fonction de ses critères. Une salle a une position globale dans le campus (cf. Fig. 1). On peut définir également une position par rapport au bâtiment d'appartenance (interface WEB). Une salle appartient à un bâtiment, et a un numéro dans ce bâtiment. Enfin, une salle peut être ouverte ou fermée.

Un bâtiment a les mêmes caractéristiques qu'une salle, mais n'a pas de type.

Une équipe est composée de membres. Elle a des besoins qu'elle exprime en nombre de pièces désirées par type, en éloignement par rapport à un lieu qui peut être un bâtiment, un éloignement par rapport à une autre équipe, et enfin en terme de dispersion. Elle affecte à chacun de ces besoins un coefficient. La somme des coefficients de chaque équipe est une constante afin qu'une équipe ne soit pas favorisée par rapport à l'autre.

On cherche à maximiser la satisfaction de toutes les équipes, connaissant leurs préférences.

Pour compléter le problème, on estime que les utilisateurs ne posent pas précisément leurs préférences. Il est possible que les préférences "déclarées" ne soient pas, consciemment ou non, les préférences réelles de l'utilisateur du fait d'une mauvaise appréciation de l'importance apportée à chaque préférence, ou encore du fait que l'utilisateur n'a pas au début une bonne connaissance de ses desiderata. Il faut donc envisager un apprentissage des préférences réelles connaissant les préférences initiales et les notes données à chaque solution.

2.2 Modélisation

2.2.1 Ajout de salles virtuelles

La méthode utilisée pour résoudre ce problème repose sur l'exécution de permutations faisant évoluer la fonction critère dans le sens désiré. Concrètement, la méthode teste toutes les permutations possibles de valeurs de deux variables et retient celle qui fait évoluer le plus la fonction critère.
Il est donc nécessaire d'ajouter au système des salles virtuelles afin de pouvoir satisfaire les contraintes de type le nombre de salles désirées. Si ces salles virtuelles n'existent pas , et dans le cas où une situation initiale n'affecte aucune salle à une équipe donnée, cette équipe n'a aucune chance d'obtenir des salles par cette méthode.
Les salles virtuelles sont affectées initialement aux équipes n'ayant pas le nombre désiré de salle dans la situation de départ. Les permutations sont ensuite étudiées sur l'ensemble des salles, virtuelles et réelles.

2.2.2 Les constantes et les variables du système

Dans la suite de ce document, afin de modéliser le problème, les variables utilisées sont, pour décrire le système :

- T : vecteur des variables du système (voir plus bas).
- N : nombre total d'équipe, n est un indice sur les équipes.
- B : le nombre de bâtiment du système, b est un indice sur les bâtiments.
- τ : nombre de type de pièce différent du système.
- k_i : nombre de pièce de type i, $i \in \{1, \tau\}$.
- K : le nombre total de pièce tel que $K = \sum_{i=1}^{\tau} k_i$.
- k^b : le total des pièces du bâtiment b tel que $k^b = \sum_{i=1}^{\tau} k_i^b$.
- P^n : le nombre total de pièces demandées par l'utilisateur n.
- P_i^n : nombre de pièces de type i demandé par l'utilisateur n.
- v_i : nombre de pièces de type i qu'il faudrait ajouter pour satifaire tout le monde.
- V : le total des pièces qu'il faudrait ajouter tel que $V = \sum_{i=1}^{\tau} v_i$. Par la suite ces pièces sont considérées comme les autres mais sont déclarées virtuelles.

- DB : matrice des distances entre les pièces et les bâtiments telle que $DB = [db_{ib}]$ où i est l'indice portant sur les pièces et b l'indice portant sur les bâtiments. Le cas particulier $db_{ib} = 0$ traduit l'appartenance de la pièce t_i au bâtiment b.

- DK : matrice des distances entre chaque piéce, par définition symétrique, de diagonale nulle.
- $TYPE$: vecteur donnant le type des salles. Mêmes indices que T. Pour les salles virtuelles t_x, $x > K$, $TYPE(x) = \tau + 1$;
- PP : Matrice des préférences telle que chaque ligne correspond à une contrainte, et chaque colonne correspond à une équipe.
- α : Matrice des coefficients associés à chaque préférence.
- Pt : Matrice des projections sur les variables. Les indices de colonnes sont ceux de T. Il y a autant de lignes que de contraintes.
- Pn : Matrice des projections sur les équipes pour chaques contraintes. Même taille que PP.
- nb_e2 : Nombre de contraintes exprimables type distance à bâtiment. Constante fixée par l'administrateur.
- nb_e3 : Idem pour les contraintes type distance à équipe.

2.2.3 La fonction objectif pour la recherche de solutions.

Le problème d'optimisation posé propose de minimiser une fonction objectif qui est l'insatisfaction des utilisateurs. Cette fonction n'est pas connue à priori car le système n'a connaissance de sa valeur que lorsque l'utilisateur note une solution. Toutefois il est possible d'estimer cette note en calculant les erreurs entre une solution et les besoins exprimés. Ce calcul est facilité par la méthode de résolution (recherche adaptative) qui cherche à minimiser ces erreurs.

Une note est d'autant plus élève que l'insatisfaction est basse. Si ERR_MAX est la plus haute note possible, alors on cherche à minimiser une fonction $J(.)$ telle que $J(.) = ERR\_MAX - note$.

Les variables du système sont les piéces à attribuer, la valeur de chacune de ces variables étant le numéro de l'équipe à qui est allouée la pièce.

Le vecteur des variables T est défini tel que

$$T = \{t_x\}$$

où $x \in \{1, K + V\}$ est l'indice de la piéce pouvant être réelle ou virtuelle.

Les variables ont le domaine de définition suivant :

$$t_x \in [0, \ldots, N],$$

La valeur 0 traduit le fait qu'une pièce n'est affectée à aucune équipe.

On note $\Phi_i(n,T)$ l'ensemble des salles réelles de type i allouées à l'équipe n. $\phi_i(n,T)$ est le nombre de pièces réelles de type i allouées à cette équipe :

$$\Phi_i(n,T) = \{t_x \mid t_x = n, x \in \{1,K\}, TYPE(x) = i\} \quad (1)$$
$$\phi_i(n,T) = Card(\Phi_i(n,T)) \quad (2)$$

La fonction objectif que nous cherchons à minimiser est $J(T)$.

2.2.4 Les contraintes.

Les contraintes du système sont connues grâce aux préférences exprimées par les utilisateurs (matrices PP et α). Elles sont de plusieurs ordres. Un utilisateur peut préciser pour son équipe un certain nombre de préférences et en omettre d'autres. Il définit également un coefficient d'importance pour chacune de ses préférences. Une préférence de coefficient nul est considérée comme non posée.

On note χ le nombre de types de préférences exprimables. Le projet s'est interessé à $\chi = 4$ types de préférences (nombre de salles désirées par type, distance à un lieu, distance à une équipe, et dispersion). Chaque équipe peut définir au maximum un nombre χ_i fixé de préférences de type i, égale pour toutes les équipes.

Les préférences posées se traduisent en contraintes sur T, vecteur des variables. Elles sont notées $C^n_{i,j}(T)$ avec $i \in \{1,\ldots,\chi\}$, $j \in \{1,\ldots,\chi_i\}$, et $n \in \{1,N\}$.

Le coefficient attribué aux préférences par une équipe n est noté $\alpha^n_{i,j}$, avec les mêmes indices que précédemment, et tel que pour une équipe n donnée : $\forall i \in \{1,\chi\}$ on a $0 \leq \alpha^n_{i,j} \leq 1$, et

$$\sum_{i=1}^{\chi} \sum_{j=1}^{\chi_i} \alpha^n_{i,j} = 1$$

Expression des préférences utilisateurs sous Scilab :

Les préférences sont modélisées comme suit :

$$PP = \begin{pmatrix} 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 1 & 1 & 1 & 7 & 1 & 1 & 7 & 1 & 1 & 1 \\ 17 & 10 & 0 & 14 & 25 & 2 & 0 & 10 & -17 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 4 & 0 & 0 & -9 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Chaque colonne représente un utilisateur (10 pour cet exemple). Chaque ligne correspond à une contrainte particulière, classée par type de haut en bas :

- De la première colonne à la colonne τ : contraintes sur les type de salles désirées, par ordre croissant de type. Ici $\chi_1 = \tau = 4$. Les valeurs sont le nombre de pièces du type demandées.
- De la colonne $\tau + 1$ à $\tau + nb_{e2}$: contraintes de type distance à bâtiment. Positif pour un rapprochement, négatif pour un éloignement. La valeur est le numéro du lieu concerné. Si la valeur est inférieure au nombre de bâtiment, le lieu est le numéro d'ordre du bâtiment à la lecture du fichier `campus.xml`. Sinon c'est un point d'interêt défini par l'equipe. Ici $\chi_2 = nb_{e2} = 2$ (lignes 5 et 6 de la matrice).
- De la colonne $tau + nb_{e2} + 1$ à la colonne $tau + nb_{e2} + nb_{e3}$: contrainte de distance à équipe. La valeur est le numéro d'ordre de l'equipe à la lecture du fichier `users.xml`. Tout comme la deuxième contrainte, la valeur est positive ou négative. Ici $\chi_3 = nb_{e3} = 2$ (lignes 7 et 8 de la matrice).
- Dernière ligne : contrainte dispersion. Si l'équipe a posé cette contrainte la valeur vaut 1, sinon 0. Dans tous les problèmes $\chi_4 = 1$, une seule contrainte de ce type est exprimable.

La matrice des coefficients associés est isomorphe à PP. Ici pour les cinq premières équipes :

$$\alpha = \begin{pmatrix} 0.1428571 & 0.1428571 & 0.1428571 & 0.1428571 & 0.1428571 \\ 0.1428571 & 0.1428571 & 0.1428571 & 0.1428571 & 0.1428571 \\ 0.1428571 & 0.1428571 & 0.1428571 & 0.1428571 & 0.1428571 \\ 0.1428571 & 0.1428571 & 0.1428571 & 0.1428571 & 0.1428571 \\ 0.1428571 & 0.1428571 & 0 & 0.1428571 & 0.1428571 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.1428571 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0.2857143 & 0.2857143 & 0.2857143 & 0.2857143 & 0.2857143 \end{pmatrix}$$

Les coefficients étant normalisés pour chaque équipe, on peut vérifier que la somme de chaque colonne vaut 1. Un coefficient nul traduit une contrainte non exprimée.

L'équipe 3 a donc demandé 5 pièces de type 1, 7 de type 2, 2 de type 3 et 1 de type 4 comme toutes les autres équipes. Mais elle a aussi demandé à être proche de l'équipe 1 avec un coéfficient de 0.14. Enfin, elle a demandé à être regroupée.

3 Méthode de recherche adaptative

La méthode de résolution utilisée est la recherche adaptative, décrite dans [2]. C'est une méthode d'optimisation combinatoire adaptée au cas étudié. Pour un problème fortement contraint pour lequel l'espace des solutions peut être vide, elle permet de calculer une solution qui tente de violer le moins possible les contraintes exprimées. C'est une méthode de recherche local.

3.1 La méthode

La première étape est d'associer à chaque contrainte une fonction d'erreur s'apparentant à une pénalisation extérieure de fonction critère. Une fonction d'erreur est nulle si la contrainte associée est satisfaite, et elle croît à mesure que la contrainte est violée.

Par exemple, soit la contrainte portant sur une variable x, vérifiée si $x < 0$. Une fonction d'erreur associée pourra être :

$$E(x) = \begin{cases} 0 \text{ si } x < 0 \\ f(x) \text{ sinon} \end{cases}$$

avec $f(x) = x$, ou $f(x) = x^2$, ou encore $f(x) = e^x - 1$.

Dans un problème disposant de plusieurs variables, on distingue une valeur d'erreur globale, et les projections de cette erreur sur les variables.

Dans le cas où le problème a plusieurs contraintes, chaque contrainte se projette sur chaque variable. On détermine alors une projection totale sur chaque variable, qui peut être une somme des projections de chaque contrainte.

L'algorithme cherche quelle variable viole le plus les contraintes. Typiquement c'est celle dont la projection totale est la plus élevée. Puis il va chercher à permuter la valeur de cette variable avec la valeur d'une autre variable. Cette dernière est déterminée en considérant, parmi toutes les permutations possibles, celle qui engendre l'erreur globale minimum.

Cette méthode pouvant conduire à des minima locaux, l'utilisation d'une liste tabou permet d'envisager des permutations sur des variables dont l'erreur projetée n'est pas maximale.

De plus, un procédé appelé "restart" permet de repartir d'une solution initiale nouvelle lorsque qu'aucune permutation améliorante n'est possible. En différenciant la situation de départ grâce à une génération aléatoire de situation, il est possible d'obtenir des solutions de moindre coût, sans être assuré cependant de l'optimalité de cette solution.

Les critères d'arrêt possibles sont :
– nombre d'itérations (permutations),

- nombre de restarts,
- évolutions du coût inférieur à un seuil.

3.2 Application au problème, notations

Afin d'appliquer cette méthode au problème posé, il est nécessaire de définir les fonctions d'erreur liées à chaques contraintes envisageables. Ces fonctions, reprises des travaux des stagiaires précédents, ont fait l'objet d'adaptation afin qu'elles soient normalisées et comparables entre elles à coefficients égaux. Etant assez libre quant au choix des fonctions d'erreur, il est possible de faire en sorte que leurs valeurs aient le même domaine de définition.
Néanmoins, compte tenu du nombre très élevé d'exécution de chaque fonction d'erreur, il est nécessaire que celles-ci soient le plus simple possible afin de nécessiter peu de calcul. A titre d'exemple, si on considère un campus de plus de 700 salles comme celui de l'INRIA, les fonctions d'erreur sont évaluées une première fois pour chaque équipe, puis pour chaque étude de permutation de salle. Cette étude de normalisation est abordée au chapitre suivant.

La fonction d'erreur associée à la contrainte $C^n_{i,j}$ est notée

$$E^n_{i,j}(T)$$

.

La projection de $E^n_{i,j}(T)$ sur une variable $t \in T$ est notée :

$$Proj_{/t}(E^n_{i,j}(T))$$

Les valeurs des fonctions d'erreur sont comprises dans un domaine tel que :

$$E^n_{i,j}(T) \in [l_i, L_i]$$

Les valeurs projetées sont elles comprises dans un domaine tel que :

$$Proj_{/t}(E^n_{i,j}(T)) \in [l'_i, L'_i]$$

3.2.1 Initialisation

Le but de l'initialisation est de fournir à l'algorithme de recherche une situation initiale. On cherche à limiter les calculs en partant d'une position aléatoire mais répondant tout de même en partie aux préférences des utilisateurs. La diversification des situations initiales est nécéssaire compte tenu du fait que la recherche adaptative est une méthode de recherche locale, cherchant systématiquement à faire baisser le coût de la fonction critère.

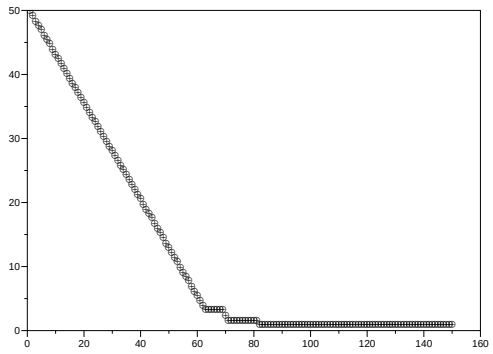
On peut imaginer plusieurs méthodes de génération de situations initiales, parmi lesquelles :

1. Situation existante (fixe).
2. Départ en salles virtuelles (fixe).
3. Départ aléatoire, avec pré satisfaction en nombre de salles (aléatoire).
4. Départ aléatoire, avec pré-répartition vis à vis des bâtiments et des équipes cibles, et pré-répartition des salles par type (aléatoire).
5. Situation existante, avec mouvements aléatoires fonctions des préférences (aléatoire).

Les trois dernières méthodes d'initialisation présentent l'avantage de pouvoir générer des situations de départ différentes sans lesquelles les restarts seraient inutiles.

La première méthode présente l'avantage de pouvoir être considérée comme une des situations les plus proches de la solution si on cherche à utiliser le logiciel pour l'optimisation d'une répartition existante. Elle perd cette caractéristique si les utilisateurs expriment des besoins qui ne correspondent pas à leur situation présente.

La deuxième méthode n'est pas optimum et part loin de la situation ce qui entraine une convergence assez lente de l'algorithme. La courbe typique de l'évolution des coûts pour un départ complet en salles virtuelles est donnée par la figure suivante :



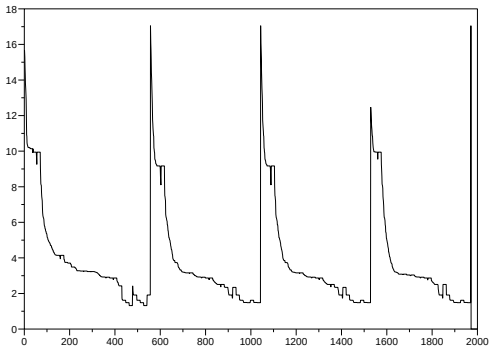
Initialisation par salles virtuelles. Pas de restart.

La longue phase de descente linéaire correspond aux permutations entre salles virtuelles et salles réelles. Cette méthode a été implémentée pour générer des situations initiales qui ont permis de valider l'implémentation de la recherche adaptative. Pour une analyse détaillée de l'affectation des salles, voir l'annexe A.

La troisième méthode est implémentée. Elle cherche à répartir les équipes aléatoirement dans les bâtiments demandés sans tenir compte du type de salles demandées. Si besoin, le

nombre de salles demandées est satisfait par un complément de salles virtuelles. Cette méthode est une amélioration de la seconde. Cependant elle ne prend pas en compte les autres desiderata des équipes, et n'approche pas suffisamment la solution.

La quatrième méthode est implémentée sous le nom d'"initialisation glouton". Cette méthode cherche à répartir au mieux les équipes en fonction de leurs demandes, sans toutefois prendre en compte la dispersion. Le caractère aléatoire est obtenu en traitant les équipes dans un ordre quelconque. Cependant, cette caractèristique peut se perdre si les équipes expriment toutes des besoins non concurents (demande de bâtiments différents). Cependant, reprenant l'hypothèse initiale de problème sur-contraint, on peut considérer que les situations générées différent à chaque fois.



Initialisation glouton, trois restarts.

La figure précédente montre que la descente vers un minimum local est plus rapide.

La cinquième méthode n'a pas été traitée, mais serait intéressante pour les mêmes raisons que la première méthode.

3.2.2 Contrainte nombre de pièces désirées

La première contrainte est liée à la différence entre le nombre de salles de type i attribuées à une équipe et le nombre de salles de même type demandées par cette même équipe. Pour cette contrainte un utilisateur peut définir autant de préférence qu'il existe de type de salle ($\chi_1 = \tau$). La contrainte est définie comme suit :

$$C^n_{1,i}(T) \leq 0 \quad \Leftrightarrow \quad P^n_i - \phi_i(n,T) \leq 0$$

Où $\phi(n,T)$ est le nombre de pièces allouées à l'équipe n par la solution T :

La fonction d'erreur associée est :

$$E^n_{1,i}(T) = \max\{0, P^n_i - \phi_i(n,T)\}$$

La valeur projetée sur les salles virtuelles est la valeur de la fonction d'erreur normalisée et coefficientée par le poids de la contrainte fixé par l'utilisateur. La valeur projetée sur les salles réelles est explicitée dans la partie "contraintes hétérogènes". La projection sur l'équipe est la somme de ces deux projections.

Implémentation Scilab :

La fonction Scilab chargée de calculer cette contrainte est `e1.sci`.

Pour toutes les fonctions d'erreur, la variable `i` est le numéro d'ordre de la contrainte (ici comprise entre 1 et τ), correspondant au type de pièce sur lequel on calcule l'erreur. La variable `n` est le numéro de l'équipe traitée.

```
1   function [Pn,Pt,Pn_val] = e1(Pn,Pt)
2       e = max(0, PP(i,n) - phit(n,i));
3
4       Ttemp  = find(T(1,1:K) == n);
5       Ttemp1 = find(TYPE ~= i);
6       Ttemp1 = intersect(Ttemp,Ttemp1);
7       Ttemp2 = [];
8
9       if (size(T,2)>K)
10          Ttemp2 = find(T(i,K+1:$) == n) + K;
11      end;
12
13      proj1 = e * Alpha(i,n) * Beta2(i,n);
14      Pt(i,Ttemp2) =  proj1;
15
16      if (phit(n,i) ~= 0) then
17          beta  = min( Beta2( find( Beta2(tau+1:$,n) > 0)+tau , n ) );
18          a     = min( Alpha( find( Alpha(tau+1:$,n) > 0)+tau , n ) );
19          proj2 = (e/phit(n,i)) * mean_min_DK * beta * a;
20          Pt(i,Ttemp1) =  proj2;
21      else
22          proj2 = 0;
23      end;
24
25      Pn(i,n) =  proj1 + proj2;
26      Pn_val = 999999999;
27  endfunction
```

Les valeurs du vecteur `phi` et de la matrice `phit`, correspondent respectivement aux nombres totaux de pièces réelles attribuées par équipe, et aux nombres de pièces réelles par type attribuées par équipe. Ces vecteurs sont calculées dans `projetesol.sce` avant le calcul de toutes les erreurs.

Le calcul de l'erreur est effectué en ligne 2. La projection sur l'équipe `n` est faite ligne 24. L'erreur, normalisée et coefficientée, est projetée sur les salles virtuelles de l'équipe (lignes 13), dont les indices sont déterminés ligne 10, si elles existent.
Les projections sur les salles sont différentes suivant que les salles sont réelles (ligne 18), ou non (lignes 13). Les valeurs projetées sur les salles réelles sont explicitées dans le chapitre "contraintes hétérogènes".

Les matrices `Beta1` et `Beta2` ont pour but de normaliser les contraintes les unes par rapport aux autres. Elles sont définies dans le chapitre 3.

La ligne 25 renseigne une valeur qui n'a pas de sens pour cette contrainte mais qui en a pour les autres. Elle est renseignée ici car c'est une valeur renvoyée par la fonction et l'appel des fonctions d'erreur est générique.

3.2.3 Contrainte distance à un bâtiment

Un utilisateur peut exprimer une préférence en terme d'éloignement à un lieu. Ce lieu peut être un bâtiment, ou un point du campus défini par l'utilisateur. Il peut également vouloir être soit au plus près, ou au plus loin. Les contraintes de ce type doivent être satisfaites dés lors que :

1. l'équipe est au plus près, voir dans, le bâtiment donné, ou au plus loin du bâtiment.
2. la distance moyenne des salles réellement attribuées au lieu donné est faible, ou grande.

Les contraintes ainsi formulées invitent à minimiser ou à maximiser une distance. Ce sont donc des fonctions objectifs à part entière. Pour les transformer en contraintes pures il faut ajouter un paramètre Δ donnant une distance en deça (ou au dela) de laquelle la contrainte sera satisfaite.

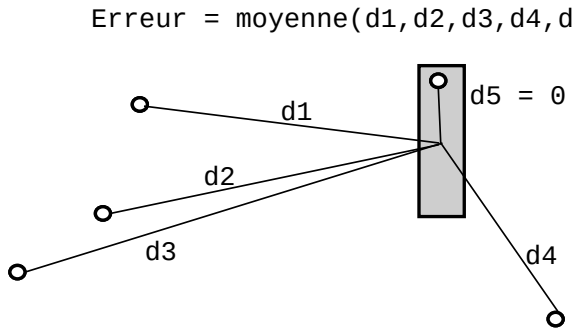
Si l'équipe n demande à être proche du bâtiment b, la contrainte peut s'écrire :

$$C_2^n(T) \leq 0 \quad \Leftrightarrow \quad \frac{\sum_{t_i \in \Phi(n,T)} db_{ib}}{\phi(n,T)} - \Delta \leq 0$$

On définit une fonction d'erreur associée à cette contrainte comme suit :

$$E_2^n(T) = \frac{\sum_{t_i \in \Phi(n,T) \cap \{t_i \notin b\}} db_{ib}}{\phi(n,T)}$$

Cette fonction est définie si $\phi(n,T) \neq 0$. Elle est d'autant plus grande que la distance moyenne des salles attribuées à l'équipe n au bâtiment b est grande. Elle est nulle si toutes les pièces attribuées sont dans le bâtiment b :



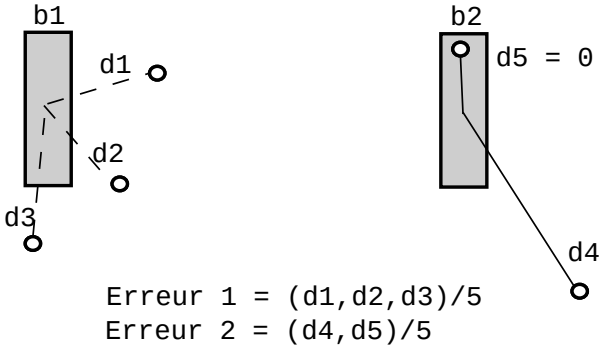
La distance $d5$ est nulle la salle est dans le bâtiment visé.

Si l'équipe souhaite au contraire être éloignée de ce même bâtiment, la fonction s'écrit :

$$E_2^n(T) = D - \frac{\sum_{t_i \in \Phi(n,T) \cap \{t_i \notin b\}} d_{ib}}{\phi(n,T)}$$

D étant la distance maximum séparant b des autres bâtiments. Ainsi, l'équipe n est d'autant plus satisfaite qu'elle se situe dans le bâtiment le plus éloigné de b.

Il est possible, en fonction des paramètres initiaux, de formuler plusieurs contraintes de ce type. Or, telle que définie ci-dessus, cette contrainte peut générer des solutions de coût minimum en affectant une équipe entre deux bâtiments dont on veut être proche. Pour corriger ce défaut, les projections sur les variables sont modifiées de telle sorte que pour chaque équipe, on ne conserve que la distance à bâtiment la moins grande. Ainsi, en demandant à être proche de $b1$ et de $b2$, l'algorithme placera l'équipe dans $b1$ et/ou dans $b2$.



L'équipe souhaite être dans $b1$ et/ou dans $b2$.

Implémentation Scilab :

La fonction Scilab chargée de caluler cette contrainte est `e2.sci`.

```
1   function [Pn,Pt,Pn_val] = e2(Pn,Pt)
2       Pn_val = 0;
3       Ttemp1 = find(T(1,1:K)==n);
4       Ttemp2 = DB(Ttemp1,abs(PP(i,n)));
5
6       if (phi(n) ~=0) then
7           if (PP(i,n)>0) then
8               Pt(i,Ttemp1) = Pt(i,Ttemp1) + Ttemp2' * Alpha(i,n) * Beta2(i,n);
9               Pn_val = sum(Ttemp2)/phi(n) * Beta1(i,n);
10          else
11              Pt(i,Ttemp1) = Pt(i,Ttemp1) + (ERR_MAX - Ttemp2' * Beta2(i,n))* Alpha(i,n);
12              Pn(i,n) = Pn(i,n) + (ERR_MAX - sum(Ttemp2)/phi(n) * Beta1(i,n))* Alpha(i,n);
13              Pn_val  = ERR_MAX - sum(Ttemp2)/phi(n) * Beta1(i,n);
14          end;
15      end;
16
17  endfunction
```

Ligne 2 : vecteur des indices des salles réelles attribuées à l'équipe 'n'.

Ligne 3 : vecteur des distances des salles réelles de l'équipe 'n' au lieu visé.

Le test ligne 6 vérifie que l'équipe a bien des salles réelles avant d'effectuer les calculs.

La projection est différente selon que l'on veut être proche ou loin d'un lieu. Cette information est donnée par le signe de la valeur du paramètre dans PP (ligne 7).

Les projections sur les variables sont réalisées aux lignes 8 et 11. Les valeurs projetées sur les salles réelles sont les distances entre ces salles et le bâtiment visé.

Dans le cas de contraintes de rapprochement, les projections sur l'équipe ne sont faites qu'après détermination de toutes les projections de cette fonction d'erreur sur les variables, afin de ne conserver que les distances à lieux les plus courtes (opération réalisée dans `projetesol.sce`). Sinon, dans le cas d'un éloignement à bâtiment, la projection sur l'équipe est faite en ligne 12.

La variable `Pn_val`, abordée dans la précédente contrainte, est la valeur de l'erreur non coefficienté. Cette valeur est utilisée par l'algorithme d'apprentissage.

3.2.4 Contrainte distance à une équipe

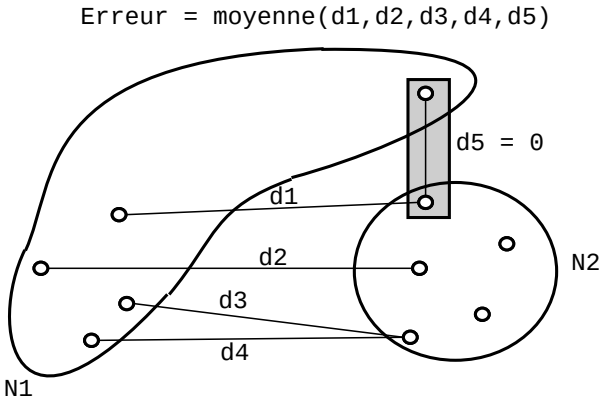
Cette contrainte est satisfaite si l'équipe qui la spécifie est proche de l'équipe précisée, ou au contraire éloignée.

Soient n_1 et n_2 deux équipes. L'équipe n_1 fixe la contrainte et demande à être proche de n_2.

L'algorithme calcule la fonction d'erreur en évaluant une variable E. Il teste si pour chaque salle t_1 attribuée à n_1 il existe une salle t_2 attribuée à n_2 dans le même bâtiment que t_1.

– Si c'est le cas E n'est pas modifiée.

– S'il n'y a pas de salle t_2 dans le bâtiment de t_1, alors E est incrémentée de la distance e minimum séparant t_1 de l'ensemble des salles de n_2. e est la valeur projetée sur la variable t_1 (normalisée, coefficientée).



La distance $d5$ est nulle car une salle de $n2$ est dans le bâtiment de $n1$.

Autrement dit :

$$E = \sum_{t_x \in \Phi(n_1,T)} e_x \quad \text{avec} \quad \begin{cases} e_x & = & 0 \text{ si } \exists b \in \{1,B\},\ \exists t_{x'} \in \Phi(n_2,T), \text{ tel que } (t_x,t_{x'}) \in Bat_b \\ & = & \min_{t_{x'} \in \Phi(n_2,T)} dk_{x,x'} \text{ sinon} \end{cases}$$

La fonction d'erreur sur cette contrainte s'écrit alors :

$$E_2^n = \frac{E}{\phi(n_1,T)}$$

Si n_1 demande à être loin de n_2 alors la fonction devient :

$$E_2^n = D - \frac{E}{\phi(n_1,T)}$$

D étant la distance maximale séparant deux bâtiments du système. Il est peu probable qu'une équipe demande à être loin d'une autre. Cependant cette possibilité a été développée en symétrie à la deuxième contrainte (distance à bâtiment).

Tout comme la contrainte précédente, la contrainte C_3^n peut être considérée comme une fonction objectif dont il faut minimiser (ou maximiser) la valeur.

Implémentation Scilab :

La fonction Scilab chargée de caculer cette contrainte est `e3.sci`.

```
1   function [Pn,Pt,Pn_val] = e3(Pn,Pt)
2       Pn_val = 0;
3       Ttemp1 = find( T(1,1:K) == n );
4       Ttemp2 = find( T(1,1:K) == abs(PP(i,n)) );
5
6       if ( Ttemp2 ~= [] ) then
7           Ttemp3 = DK(Ttemp1,Ttemp2).*MMK(Ttemp1,Ttemp2);
8           Ttemp  = min(Ttemp3,'c');
9
10          if (phi(n) ~= 0) then
11              if (PP(i,n) > 0) then
12                  Pt(i,Ttemp1) = Ttemp'           * Alpha(i,n) * Beta2(i,n);
13                  Pn(i,n)      = sum(Ttemp)/phi(n)* Alpha(i,n) * Beta1(i,n);
14                  Pn_val       = sum(Ttemp)/phi(n)*Beta1(i,n);
15              else
16                  Pt(i,Ttemp1) = (ERR_MAX - Ttemp'           * Beta2(i,n))* Alpha(i,n);
17                  Pn(i,n)      = (ERR_MAX - sum(Ttemp)/phi(n)* Beta1(i,n))* Alpha(i,n);
18                  Pn_val       =  ERR_MAX - sum(Ttemp)/phi(n)* Beta1(i,n);
19              end;
20          end;
21
22      end;
23
24  endfunction;
```

$Ttemp1$ et $Ttemp2$ sont les vecteurs des indices des salles réelles attribuées respectivement à l'équipe n et à l'équipe visée par n (lignes 3 et 4). Le calcul n'est fait que si l'équipe visée a des salles réelles (ligne 6), les projections étant nulles sinon.

$Ttemp3$ est la matrice des distances entre les salles affectées à l'équipe n vers les salles affectées à l'équipe visée par n. Cette distance est nulle lorsque deux salles de chaque équipe sont dans le même bâtiment. La matrice carrée MMK est définie telle que $mmk_{ij} = 0$ si deux salles sont dans le même batiment, et $mmk_{ij} = 1$ sinon.

Au final, et conformément à la définition de la fonction d'erreur, on ne conserve que les distances minimales de chaques salles de l'équipe n aux salles de l'équipe visée (ligne 8).

Ensuite, de la même façon que pour la deuxième contrainte, ces valeurs sont projetées sur les variables correspondantes, et leur moyenne est projetée sur l'équipe. La projection n'est faite que si n a des salles réelles, et en fonction du critère d'éloignement ou de rapprochement (signe du numéro de l'équipe cible). L'ensemble de ces valeurs sont normalisées par $Alpha$, $Beta1$ et $Beta2$.

3.2.5 Contrainte dispersion d'une équipe

L'objectif de cette contrainte est de favoriser le rapprochement des salles attribuées à une équipe sans tenir compte du bâtiment. Tout comme les deux contraintes précédentes, il faut définir un seuil Δ pour exprimer cette contrainte. Si $\Phi(n,T)$ est l'ensemble des salles attribuées à l'équipe n :

$$C_4^n(T) \leq 0 \leftrightarrow \frac{\sum_{t_x \in \Phi(n,T)} \max_{t_{x'} \in \Phi(n,T)} dist(t_x, t_{x'})}{\phi(n,T)} - \Delta \leq 0$$

La fonction d'erreur associée est :

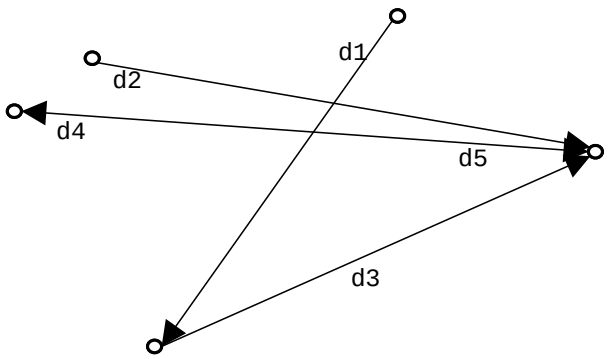
$$E_4^n(T) = \frac{\sum_{t_x \in \Phi(n,T)} \max_{t_{x'} \in \Phi(n,T)} dist(t_x, t_{x'})}{\phi(n,T)}$$

La valeur projetée sur la salle t_i réelle est la distance maximale séparant cette salle des autres salles de l'équipe :

$$Proj_{/t_x}(E_4^n(T)) = \max_{t_{x'} \in \Phi(T)} dist(t_x, t_{x'})$$

Clairement, cette contrainte est prise en compte seulement si $\Phi(n,T) \geq 2$. Sinon, l'erreur est nulle.

```
Erreur = moyenne(d1,d2,d3,d4)
```



La distance $d4$ est le diamètre de l'équipe car $= \max\{d1, d2, d3, d4, d5\}$.

Cette contrainte a évoluée par rapport aux versions précédentes du logiciel suite à l'étude de normalisation des contraintes hétérogènes menée plus loin. Initialement, la fonction d'erreur avait pour valeur le rayon maximum de l'équipe, c.à.d la distance maximale séparant deux salles réelles.

Implémentation Scilab :

La fonction Scilab chargée de caluler cette contrainte est `e3.sci`.

```
1  function [Pn,Pt,Pn_val]=e4(Pn,Pt)
2
3      Ttemp1 = find(T(1,1:K) == n);
4      Ttemp2 = max(DK(Ttemp1,Ttemp1),'r');
5      Pt(i,Ttemp1) = Ttemp2             * Alpha(i,n) * Beta2(i,n);
6      Pn(i,n)      = sum(Ttemp2)/phi(n) * Alpha(i,n) * Beta1(i,n);
7      Pn_val       = sum(Ttemp2)/phi(n) * Beta1(i,n);
8
9  endfunction
```

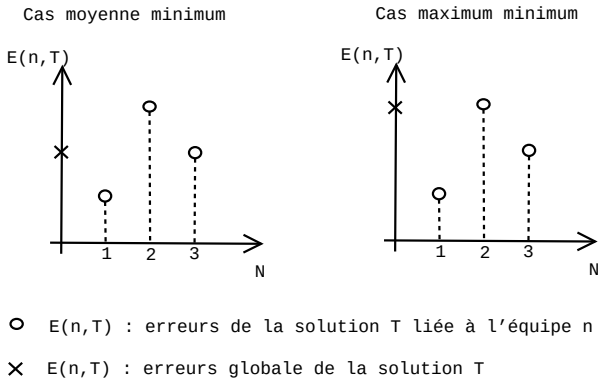
Aprés avoir extrait les indices des salles réelles attribuées à l'équipe n (ligne 3), et déterminé les distances maximales séparant chacunes de ces salles aux autres de l'équipe (ligne 4), la projection de ces valeurs maximales et de leur moyenne sont projetées respectivement sur les variables et sur l'équipe (ligne 5 et 6), aprés les normalisations.

3.2.6 Fonction objectif

On définit la projection globale des erreurs sur une équipe comme étant la somme des projections de chaque fonction d'erreur sur cette même équipe. Du fait de la définition des coefficients α, et de la normalisation des projections (voir plus bas), cette valeur est comprise entre 0 et ERR_MAX. Plus cette valeur est basse, plus l'équipe est satisfaite. L'objectif est de chercher le minimum "consensuel" de l'insatisfaction de chaque équipe.

Les premières implémentations du logiciel considéraient la moyenne de ces valeurs comme fonction objectif à minimiser. L'étude montre que cette fonction n'est pas satisfaisante (cf. annexes). Elle autorise en particulier la complète satisfaction des équipes demandant le moins de salles, à préférences égales. Une moyenne basse ne garantie pas l'insatisfaction consensuelle minimale de chaque équipe.

Il est possible de corriger les fonctions d'erreur de façon à modifier certaines répartitions, au prix cependant d'une complexité algorithmique supplémentaire, pénalisante en temps de calcul. C'est pourquoi la fonction critère a évoluée et on cherche maintenant à minimiser l'insatisfaction maximum. L'étude rapportée en annexe montre que avec cette seule modification, et à coefficients égaux, la contrainte n°1 permet une juste répartition des salles entre les différentes équipes.



Moyenne minimale contre maximum minimal.

Cela implique une petite adaptation de l'algorithme de recherche adaptative. On ne considère plus la variable de coût maximum par rapport à l'ensemble des autres variables, mais la variable de coût maximum par rapport à l'ensemble des variables de l'équipe de coût maximal. Une permutation est considérée comme améliorante si elle fait baisser le coût de l'équipe concernée qui est par définition la fonction objectif. A noter que l'échange peut entraîner l'augmentation des erreurs sur les autres équipes, qui ne doivent toutefois pas dépasser l'erreur maximale.

3.2.7 Listes tabous

Le logiciel utilise deux listes tabous. La première stocke les indices des salles de grande erreur pour lesquelles aucune permutation améliorante n'existe.

La seconde stocke les indices des équipes pour lesquelles toutes les salles ont été déclarées tabous. Si une équipe est déclarée tabou deux fois avec le même coût final, alors elle est déclarée tabou définitivement. On évite ainsi de parcourir plusieurs fois les salles d'une équipe dont on sait qu'elle n'a plus de permutation améliorante possible, malgré une erreur globale maximale par rapport aux autres. Ce peut être le cas, par exemple, d'une équipe demandant la proximité d'un bâtiment et d'une équipe, les deux 'cibles' étant éloignées l'une de l'autre.

Lorsque toutes les équipes sont déclarées tabous, la recherche s'arrête et repart éventuellement d'une situation initiale différente de la précédente (restart).

3.3 Normalisation des contraintes hétérogènes

Il s'agit de determiner des coefficients de normalisation afin que les valeurs des fonctions d'erreur évoluent dans des domaines comparables. Les valeurs des fonctions d'erreur et valeurs projetées sont comprises dans des domaines tels que :

$$E^n_{i,j}(T) \in [l_i, L_i]$$

et

$$Proj_{/t}(E^n_{i,j}(T)) \in [l_i^{'}, L_i^{'}]$$

On vérifiera en particulier que $l_i = l_i^{'} = 0 \quad \forall i$ et on cherchera des coefficients β_i^1 et β_i^2 tels que :

$$L_i.\beta_i^1 = L_i^{'}.\beta_i^2 = ERR\_MAX$$

ERR_MAX est une borne d'erreur maximale fixée, inférieure aux capacités des types de variables employées dans le langage de programmation utilisé.

3.3.1 Contrainte nombre de pièces désirées

On ne traite ici que de la normalisation des projections sur les salles virtuelles. (Cf. plus bas : contraintes hétérogènes).

Le domaine de définition de cette fonction est indépendante du type de salle demandée. Le type de salle n'est donc pas une grandeur conservée pour la suite.

On considère que la contrainte est satisfaite dés lors que le nombre de salles attribuées est supérieur ou égale au nombre de salles demandées.

Le domaine des valeurs de la fonction d'erreur est $[0, P^n]$, d'où :

$$\beta_1^1 = \beta_1^2 = \frac{ERR\_MAX}{P^n}$$

Cette valeur est fonction des demandes de l'équipe n. Elle est calculable avant l'optimisation. Pour chaque équipe, il y a autant de coefficients de normalisation que de types de salle.

3.3.2 Contrainte distance à un bâtiment

Soit b le bâtiment dont on cherche à minimiser l'éloignement. Pour alléger les notations on note simplement $\phi = \phi(n, T)$, le nombre total de salles réelles attribuées à l'équipe n tous types confondus.

L'expression de l_2 s'obtient en considérant le cas où $\phi < k^b$, i.e. le nombre de salles attribuées à l'équipe n par la solution T est inférieur à la capacité du bâtiment. Dans ce cas, par définition, l'erreur est nulle. Donc :

$$l_2 = 0$$

De même on peut exprimer la borne supérieure du domaine de définition de la fonction d'erreur en considérant la salle la plus éloignée du bâtiment visé :

$$L_2 = \max_{i \in \{1,\dots,K\}} db_{ib}$$

Cette valeur ne dépend que de b. C'est un paramètre du système calculable avant l'optimisation.

Donc, l'ensemble des valeurs de la fonction d'erreur n°2 n'est pas vide. L'uniformisation avec les autres fonctions d'erreur implique :

$$\beta_2^1 = \frac{ERR\_MAX}{\max_{i \in \{1,\dots,K\}} db_{ib}}$$

Le dénominateur est la distance maximale séparant une salle du système du bâtiment visé.

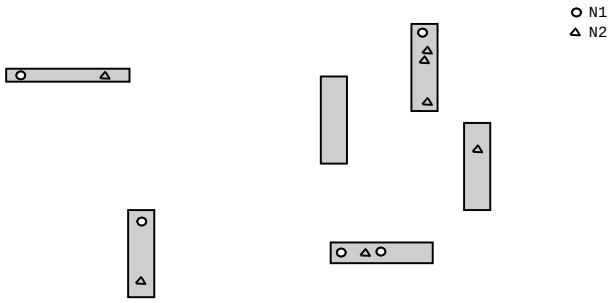
La valeur projetée sur chaque variable t_i désignant une salle réelle est la distance de la salle au bâtiment visé. Aucune valeur n'est projetée sur les salles virtuelles. Le domaine de définition de cette valeur est le même que celui de la fonction d'erreur globale. Donc :

$$\beta_2^{'} = \beta_2$$

3.3.3 Contrainte distance à une équipe

Soient deux équipes n_1 et n_2. La première souhaite la proximité de la deuxième. Par définition, la fonction d'erreur s'annule dans le cas où dans chaque bâtiment occupé par n_1 il y a au moins une salle attribuée à n_2. Cette situation est possible, et on vérifie alors :

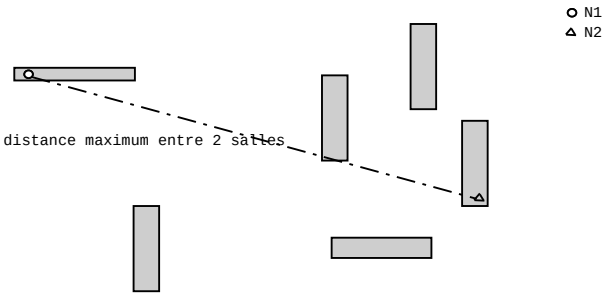
$$l_3 = 0$$



Cas d'erreur minimum.

Dans les cas particuliers $\phi(n_1, T) = \phi(n_1, T) = 0$ l'erreur est considérée comme nulle.

La valeur maximale de E_3^n s'obtient si $\phi(n_1, T) = \phi(n_2, T) = 1$ et si les salles attribuées à n_1 et n_2 sont les plus éloignées l'une de l'autre dans le système.



Cas d'erreur maximum.

$$L_3 = \max_{(i,j)\in\{1,\ldots,K\}^2} dist(t_i, t_j)$$

La valeur obtenue est indépendante de n_1 et de n_2. On obtient une valeur constante, caractérisant le système, assimilable au diamètre du campus.

Le domaine de définition de E_3^n n'est donc pas vide, et admet une borne supérieure fixe. Il est alors possible de définir le coefficient de normalisation β_3^1 de la fonction d'erreur E_3^n :

$$\beta_3^1 = \frac{ERR\_MAX}{\max_{(i,j)\in\{1,\ldots,K\}^2} dist(t_i, t_j)}$$

La valeur projetée sur une salle t_i réelle est la distance minimum séparant cette salle des salles de l'équipe cible. Le domaine de cette valeur est le même que celui déterminé ci-dessus :

$$\beta_3^{'} = \beta_3$$

3.3.4 Contrainte dispersion d'une équipe

Les domaines de définitions des valeurs des projections sur équipe et sur variables sont les mêmes du fait de la définition de la fonction. La borne inférieure est atteinte pour $\phi(n, T) \leq 2$:

$$l_4 = 0$$

Quant à L_4 c'est la distance maximale séparant deux salles :

$$L_4 = L_4^{'} = \max_{(t_i,t_j)\in\{0,\ldots,K\}^2} dist(t_i,t_j) = L_3$$

L_4 est le diamètre du système.
De là on déduit :

$$\beta_4^1 = \beta_4^2 = \frac{ERR\_MAX}{\max_{(t_i,t_j)\in\{0,\ldots,K\}^2} dist(t_i,t_j)}$$

3.4 Les contraintes en concurrence

L'annexe B montre que le comportement des fonctions d'erreur lorsqu'elles sont mises en concurrence, ce pour les deux fonctions objectifs utilisées ('moyenne minimale', 'maximum minimum'). On peut en déduire certaines modifications qui permettent d'obtenir des résultats cohérents. Les fonctions d'erreurs décrites ci-dessus tiennent compte de ces remarques.

3.4.1 La fonction objectif

L'adoption de la nouvelle fonction objectif permet de résoudre le problème lié à la contrainte portant sur le nombre de salles désirées. Alors que la version 'moyenne minimum' satisfait en priorité les équipes qui demandent le moins de salles, la version 'maximum minimum' assure une bonne répartion entre les équipes à coefficients d'importance égaux.

Dans le premier cas la fonction objectif est d'autant plus basse en moyenne qu'un grand nombre d'équipe est satisfait. Résultat atteint en satisfaisant ceux qui demandent le moins. En revanche, la version 'maximum minimum' va chercher un taux d'erreur équivalent entre toutes les équipes.

3.4.2 Plusieurs contraintes d'un même type

L'annexe A montre comment se comporte l'algorithme de résolution lorsqu'il ne prend en compte qu'une contrainte de chaque type. La résolution réelle du problème suppose cependant l'expression de plusieurs contraintes de même type. Il est nécessaire de considérer le comportement des contraintes de même type les unes par rapport aux autres.

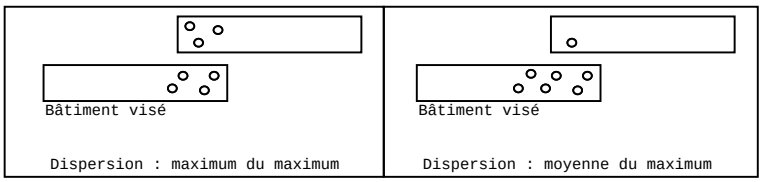
La conclusion de l'étude porte sur les contraintes de type NSD et de type distance à lieu. Les autres contraintes ne sont pas concernées. Le cas NSD est traité en détail plus bas, et a conduit à la determination des valeurs des fonction d'erreur à projeter sur les salles réelles.

Le deuxième cas est expliqué dans la définition de la fonction plus haut. Il s'agit d'éviter que dans le cas où l'utilisateur demande la proximité de deux bâtiments, le système le positionne à mi-chemin de ces deux bâtiments.

3.4.3 Projections sur équipes : unicité de la mesure

Ce principe est illustré par la quatrième contrainte, portant sur la dispersion des équipes. Initialement la valeur projetée sur l'équipe était la distance maximale séparant deux salles de l'équipe, autrement dit le diamètre de l'équipe. Ainsi définie cette contrainte prenait le pas sur les autres contraintes de type distance (2 et 3), qui sont des moyennes de distances. De plus considérant deux solutions pour lesquelles les diamètres de l'équipe sont égaux, une solution présentant une meilleur dispersion projetait la même valeur sur l'équipe que l'autre.

La version finale de la quatrième contrainte projette la valeur moyenne des distances maximale sur l'équipe.



Contrainte dispersion et contrainte distance à bâtiment

3.4.4 Réciprocité de la contrainte distance à équipe

La contrainte de distance à équipe influe sur l'équipe demandeuse n_1, en favorisant le rapprochement de n_1 avec n_2, ce qui est souhaitable. Elle influe également sur l'équipe n_2. En effet, afin de satisfaire la contrainte posée par $n1$, n_2 peut se voir affecter des salles dans les mêmes bâtiments que n_1 alors que n_2 n'a posée aucune contrainte en ce sens. On aurait pu s'attendre à ce que n_1 seule soit concernée par ce rapprochement.

Ce comportement est une conséquence de la modélisation. Il est jugé correct et laissé tel quel. Si une équipe demande la proximité à une autre c'est qu'elle a des intérêts avec elle, qui ont peut-être échappés à l'autre équipe.

Toutefois, pour minimiser la réciprocité de la contrainte et son impact sur l'équipe cible, la valeur projetée sur une salle affectée à n_1 n'est pas la distance moyenne entre cette salle et les autres salles de n_2, ce qui aurait pour effet de rapprocher les centres de gravité des deux équipes, mais la distance minimale. On considère que n_1 est satisfaite si elle dispose d'un "ambassadeur" de n_2 dans chacun de ses bâtiments.

3.4.5 Contraintes hétérogènes

Si la comparaison et la normalisation est possible entre les contraintes de type distance, elle est plus délicate entre celles de type nombre de salles désirées et celles de type distance.

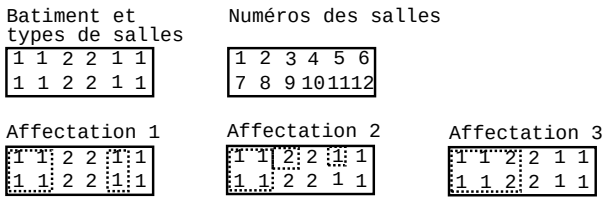
Dans le cas particulier d'un système ne disposant que d'un seul type de salle, la cohabitation des deux types de contrainte ne pose pas de problème. La contrainte type nombre de

salles désirées (NSD) entraîne des projections seulement sur les salles virtuelles en cas d'insuffisance. Lorsqu'il n'y a plus d'échanges améliorants sur le NSD, ces variables deviennent tabous et l'optimisation continue pour les autres contraintes. Dans ce cas la liste tabou et les salles virtuelles se chargent de régler le problème de l'hétérogénéité des contraintes.

De fait, on peut considérer que les deux types de contraintes évoluent indépendamment l'une de l'autre. La satisfaction en nombre de salle ne se préoccupe pas du positionnement des salles dans le campus et tant qu'il reste des salles, il est possible de faire baisser la fonction d'erreur associée.

En revanche, lorsqu'il y a plus d'un type de salle, la situation change car l'erreur liée au manque d'un type de salle particulier se projette sur les salles virtuelles comme avant, mais aussi sur les salles réelles de types différents, afin de susciter un échange avec le type de salle déficitaire. Les tests effectués ont mis en évidence la nécéssité de projeter des valeurs différentes sur les salles virtuelles et sur les salles réelles, car sinon la contrainte NSD prend le pas sur les autres contraintes. On obtient alors, à préférences égales, une satisfaction complète en NSD par type, mais aussi une dispertion non négligeable.

Ce phénomène s'illustre avec l'exemple suivant :



Caractéristiques d'un bâtiment, et répartions envisagées.

Si on considére un système disposant d'un bâtiment comptant 8 salles de type 1 et 4 de type 2. Une équipe demande à avoir 6 pièces de type 1 (poid α_1), à être dans ce bâtiment (poid α_2), et à être regroupée (poid α_4).

Compte tenu de la place disponible, considérons la contrainte distance à bâtiment satisfaite.

Le cas $\alpha_4 = 0$ doit en toute logique mener à une situation où l'ensemble des contraintes sont satisfaites, et où les fonctions d'erreur sont nulles. L'affectation 1 est un exemple de répartition à coût nul pour l'équipe.

Le cas $\alpha_4 = \alpha_1 = 0.5$ doit mener à une répartion permettant le viol de la première contrainte. L'affectation 2 est une solution particulière de ce cas.

Calculons les fonctions d'erreur dans le cas où pour celle liée au NSD, on projette la même valeur sur les salles réelles et virtuelles.

Considérant la distance de Manhatan, une erreur maximale $ERR\_MAX = 100$, et une distance maximale entre bâtiment $max_{bd} = 100$, le calcul des fonctions d'erreur pour l'affection 2 est le suivant :

$$E^1_{1,1}(T_2) = \max\{0, P^n_1 - \phi_1(n,T)\}.\beta_1.\alpha_1 = 1.\frac{100}{6}.\alpha_1$$

$$E^1_4(T_2) = \frac{\alpha_4.\beta_4}{\phi(n,T_2)} \sum_x \max_{x'} dist(t_x, t_{x'})$$

L'application numérique donne $E^1_{1,1}(T_2) = 8.33$ et $E^1_4(T_2) = 2$. La somme des erreurs projetées sur les variables sera maximale pour la salle d'indice 3 et vaut :

$$E^1_{1,1}(T_2) + Proj_{/t_3}(E^1_4(T_2)) = 9.66$$

Le coût global est le maximum des sommes des projections de ces fonctions d'erreur sur les équipes. Soit, pour la seule équipe en jeu :

$$J(T_2) = E^1_{1,1}(T_2) + E^1_4(T_2) = 10.66$$

Pour l'affectation 1, ces erreurs valent : $E^1_{1,1}(T_1) = 0$ et $E^1_4(T_1) = 2.33$. D'où :

$$J(T_1) = E^1_{1,1}(T_1) + E^1_4(T_1) = 2.33$$

Clairement, l'affectation 1 est de coût global inférieur à l'affection 2. L'objectif est d'obtenir un coût minimum pour l'affectation 2. Pour cela, il faut au moins satisfaire la condition suivante :

$$J(T_1) > J(T_2) \leftrightarrow E^1_{1,1}(T_2) \leq E^1_{1,1}(T_1) + E^1_4(T_1) - E^1_4(T_2)$$

T_1 satisfait la contrainte NSD, donc on cherche à satisfaire :

$$E^1_{1,1}(T_2) \leq E^1_4(T_1) - E^1_4(T_2)$$

Ce qui se traduit par une inégalité sur β'_1, nouveau coefficient de normalisation sur les salles réelles, pour les contraintes NSD :

$$\beta'_1 \leq \frac{\beta_4.\alpha_4}{\alpha_1.\phi(n,T_2)} * (\sum_x (\max_{x'} dist(t^1_x, t^1_{x'}) - \sum_x (\max_{x'} dist(t^2_x, t^2_{x'}))$$

On peut répéter ce même calcul avec les contraintes de type 2 et 3 (resp. distance à bâtiment et distance à équipe). Nous obtenons des inégalités de mêmes formes pour l'échange d'une salle.

Au final, on obtient une approximation générale de β'_1, coefficient de normalisation appliqué aux projections de la fonction d'erreur type NSD pour le type i sur les salles réelles, avec l'égalité suivante :

$$\beta_1' = \frac{\overline{\underline{dk}}.\beta_{min}.\alpha_{min}}{\phi_i(n,T).\alpha_1^i}$$

Avec :
- $\overline{\underline{dk}}$ moyenne des distances minimales séparant une salle des autres salles (typiquement, distance moyenne entre deux salles voisines).
- β_{min} valeur minimale des coefficients de normalisation portant sur les fonctions d'erreur de type distance.
- α_{min} valeur minimale des coefficients de préférence portant sur les fonctions d'erreur de type distance.

L'utilisation de la moyenne des valeurs minimales pour $\overline{\underline{dk}}$ est expérimentale. On peut également tester le comportement de la fonction d'erreur pour le minimum des minima, ou encore le maximum des minima, ou toute autre valeur entre ces deux extrêmes paramétrée par un coefficient.

Les essais avec la moyenne des minima montrent de meilleurs regroupements des équipes, et des viols des contraintes NSD par l'affection de salles de types non demandés.

On vérifie que $E_1.\beta_1' \leq ERR\_MAX$.

Il faut noter que la nécessaire modification des projections sur les salles réelles pour cette contrainte engendre des délais de calculs supplémentaires.

3.5 Améliorations envisageables

3.5.1 Voisinage d'approche

On peut chercher à accélerer la convergence de la recherche de solution en adoptant au début un voisinage particulier qui permettrait des échanges par blocs et non unitaires comme le fait la recherche adaptative. Ce voisinage serait un premier étage de la résolution, chargé de livrer au deuxième étage une situation moins grossière que celle générée initialement.

Un voisinage possible est de considérer les échanges entre équipes entières, une équipe n_1 échangeant ses salles avec l'équipe n_2 dont la répartition minimise les contraintes de n_1. Tout comme l'apprentissage des solutions il faut alors calculer toutes les fonctions d'erreur vis à vis de toutes les contraintes possibles, ce avant chaque échange. Le coût en calcul est important, mais doit être comparé à ce que ferait la recherche adaptative pour les mêmes permutations.

3.5.2 Prise en compte de la géométrie du campus

L'exploration des permutations possibles porte sur l'ensemble des variables du système, même celles dont on sait qu'elles ne peuvent satisfaire les contraintes.

Il doit être possible de limiter les recherches en prenant en compte la géométrie du système. Sachant par exemple qu'une équipe souhaite être proche du bâtiment b, on peut envisager de limiter la recherche aux salles proches de ce bâtiment. En procédant de même pour les autres contraintes, le domaine des salles "permutables" possibles pourraît être réduit d'autant.

Cependant, cette variante va dans le sens d'une plus grande complexité algorithmique et il faudrait considérer les gains possibles en temps de calcul.

Si certains domaines peuvent être calculés à l'avance en fonction des préférences posées, certains autres ne peuvent être déterminés qu'au moment du calcul (distance à équipe, dispersion).

4 Apprentissage des préférences utilisateurs

4.1 Problème

Le système doit s'adapter au fait que l'utilisateur n'exprime pas forcément correctement ses préférences, ou le fait de façon incomplète. D'autre part, les coefficients donnés à chaque préférences sont approchés et peuvent ne pas correspondre exactement à l'idée de celui qui les pose.

Le problème est donc d'évaluer les préférences réelles des utilisateurs au travers des préférences initialement exprimées et des notes données aux solutions calculées. On cherche ainsi à calculer de nouveaux jeux de préférences qui correspondent mieux aux notes données.

4.2 Modélisation en programmation linéaire

On ne considère plus ici un nombre limité d'erreurs, mais l'ensemble des erreurs possibles pour chaque type de contrainte, sauf pour la contrainte de type 1 (nombre de salles désirées, ou NSD), pour laquelle il y aurait trop de possibilités (autant que de nombre de salles possibles par type). Pour une équipe donnée, et pour une solution s donnée, il y aura donc une erreur $E^s_{2,+l1}$ de rapprochement à lieu, ce pour chaque bâtiment, et pour chaque lieu. De même il y aura $E^s_{2,-l1}$ pour l'éloignement à lieu. De la même façon, il faut considérer les erreurs $E^s_{+3,e1}$ et $E^s_{3,-e1}$ pour chaque équipe. En revanche, il n'y a qu'une contrainte E^s_4 de dispersion.

Nous obtenons donc une matrice F telle que (s'il n'y a pas de point d'intèrêt) les erreurs liées à la solution s soit la ligne s de F :

$$
\begin{aligned}
F^s \quad = \quad & [F^s_{1,t_1}, F^s_{1,t_2}, \ldots, F^s_{1,t_\tau}, && (3)\\
& F^s_{2,+l_1}, F^s_{2,-l_1}, F^s_{2,+l_2}, F^s_{2,-l_2}, \ldots, F^s_{2,+l_B}, F^s_{2,-l_B}, && (4)\\
& F^s_{3,+e_1}, F^s_{3,-e_1}, F^s_{3,+e_2}, F^s_{3,-e_2}, \ldots, F^s_{3,+e_N}, F^s_{3,-e_N}, && (5)\\
& F^s_4] && (6)
\end{aligned}
$$

avec :

$$F^s_* = E^s_* . \beta^n_*$$

On distingue la note donnée par l'utilisateur de la note réelle. La première étant une estimation de la seconde par l'utilisateur. On définit alors les deux vecteurs suivants :

- *Notes* : vecteur des notes données par l'utilisateur à chaque solution.
- δ : vecteur qui donne pour chaque solution l'écart entre la note réelle et la note donnée par l'utilisateur.

On considère que la note réelle d'une solution donnée par l'utilisateur est une combinaison linéaire de toutes les erreurs engendrées par la solution.

En utilisant les notations définies précédemment, on peut écrire :

$$F.\hat{\alpha} = Notes + \delta$$

où $\hat{\alpha}$ est le vecteur des coefficients réels à estimer.
Il est alors possible de calculer $\hat{\alpha}$ en appliquant la programmation linéaire au problème d'optimisation suivant :

$$\begin{cases} \min \sum_i |\delta| \\ F.\hat{\alpha} - \delta = Notes \end{cases}$$

4.3 Concaténation

Le vecteur $\hat{\alpha}$ obtenu précédemment n'est pas appliqué tel quel. Seuls les χ_2 et χ_3 contraintes de plus forts coefficients sont conservés. On ajoute ensuite les paramètres et les coefficients correspondants à ceux définis initialement en évitant les doublons et les contraintes incohérantes (rapprochement et éloignement d'un même bâtiment, distance par rapport à soit même). La dernière étape consiste à normaliser les coefficients, ce qui a pour effet de baisser les coefficients initiaux. Pour un exemple d'apprentissage, voir l'annexe D.

4.4 Conclusion

L'algorithme d'apprentissage par la programmation linéaire donne de bons résultats. Dans l'exemple donné en annexe, les solutions fournies sont cohérentes dés la première exécution de l'algorithme. La deuxième itération a permet de dégager une tendance nette vers le résultat recherché.

Cependant, l'exemple est le résultat d'un test de l'algorithme d'apprentissage réalisé en boucle. Les notes de l'utilisateur sont calculées exactement à la fin de chaque recherche de solution. A ce titre on peut vérifier que le vecteur δ est nul à l'issue de l'exécution de la programmation linéaire.
Il faut maintenant voir comment se comporte cet algorithme en présence de noteurs humains qui n'évaluent pas aussi précisément les solutions que peut le faire l'algorithme de test utilisé ici.

Enfin, cet algorithme impose le calcul de toutes les fonctions d'erreur possibles liées à chaque solution. D'ordinaire, seules les fonctions d'erreur liées aux préférences utilisateur sont évaluées lors de la recherche de solution. Pour l'apprentissage, il faut notamment évaluer les erreurs liées à la proximité et à l'éloignement de chaque bâtiment et équipe, et éventuellement celle liée à la dispersion si tel n'était pas le cas. Sinon, les coefficients obtenus ne portent que sur les erreurs liées aux préférences utilisateur. Ce point engendre des

calculs non négligeables, et augmente la taille des données sauvegardées. La sauvegarde est nécessaire car si l'algorithme d'apprentissage s'exécute avant la recherche de solution, les données qu'il utilise sont calculées aprés la recherche de solutions.

5 Consensus version 3, implémentation.

Cette partie décrit l'architecture générale de la version 3 de `consensus` ainsi que les choix faits en matière d'implémentation. Cette version est appelée 3 car elle vient après les implémentations de Mathieu Pernollet et Martin Pierres. Il ne s'agit pas pour autant d'une version opérationnelle tant la partie résolution mériterait d'être plus robuste, et l'interface mieux travaillée. Son numéro devrait plutôt être v0.3.

5.1 Définitions, l'exécution par sessions et par phases.

Une session désigne l'ensemble des opérations visant à répartir les ressources du système. Elle comprend l'initialisation du système, avec notamment la désignation des équipes actives et des bâtiments ouverts. Elle comprend ensuite un nombre variable de jeux.

Un jeu comprend deux phases :

1. notation des précédentes solutions par l'utilisateur, et suivant les cas expression de préférences initiales ou nouvelles,
2. apprentissage éventuel des préférences, puis calcul de solutions.

Il y a autant de jeux que nécéssaire.

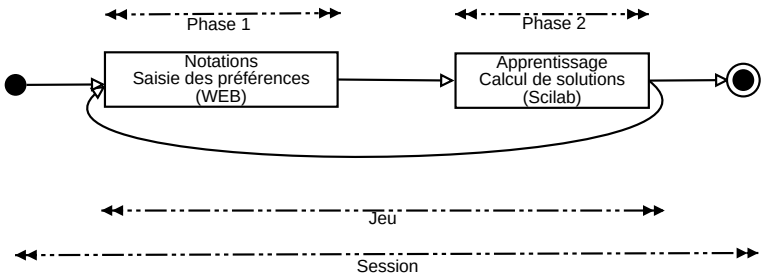
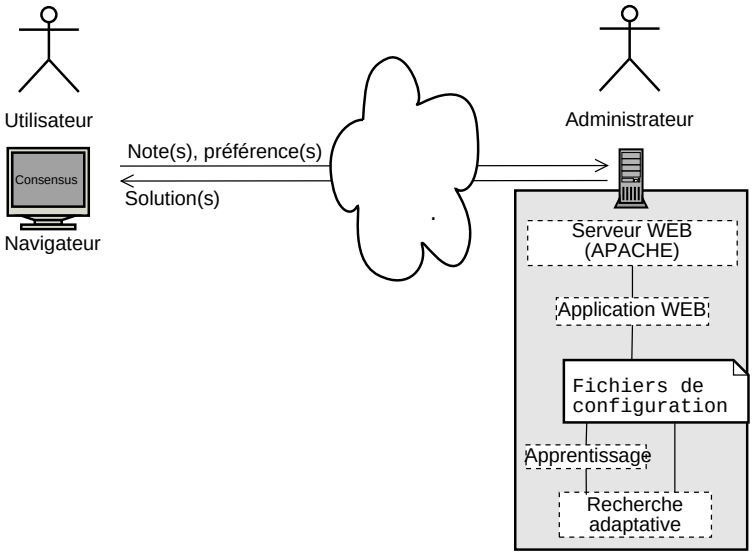


Schéma global

La succession des phases est régie par l'administrateur. Cette succession est transparente pour l'utilisateur qui peut continuer à visualiser les solutions, noter, ou poser de nouvelles préférences pendant la phase calculatoire. Cependant, ces nouvelles entrées ne seront pas prises en compte avant la prochaine phase de calculs. Une amélioration possible du logiciel consisterait à prévenir par mail l'ensemble des équipes joueuses lorsque de nouvelles solutions sont disponibles. Les temps de calculs étant long (une à plusieurs heures), il n'est pas envisageable que les utilisateurs restent attentifs pendant la totalité d'une session.

5.2 Architecture générale, choix des langages.

Pour la première phase, l'idée est que les utilisateurs puissent faire leurs choix et leurs notations d'un lieu distant de l'ordinateur chargé du calcul des solutions. Le choix a été fait de développer un service WEB, qui présente l'avantage pour le client (utilisateur) de ne nécessiter aucun autre logiciel spécifique que son navigateur WEB, qui doit cependant être au moins graphique. L'architecture globale est donc de la forme client léger, serveur lourd. De fait, la machine serveur doit être capable : de mettre à disposition un serveur WEB, de générer dynamiquement les pages WEB à partir des fichiers de configuration, et de réaliser les calculs de solutions. Il est possible de délocaliser la partie calcul de solutions sur une autre machine, les échanges étant réalisés par de simples transferts de fichiers.



Déploiement

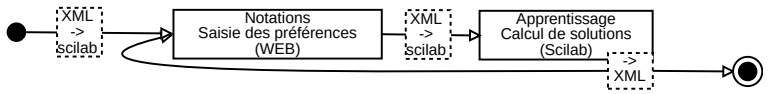
Le développement concerne donc uniquement le coté serveur. Le serveur WEB utilisé pour le développement est `APACHE`. Il permet le lancement de `CGI` en `Perl` et l'authentification par mot de passe. Ce dernier point est important car les `CGI` ne connaissent les utilisateurs que par l'intermédiaire de la variable globale `REMOTE_USER` renseignée par le serveur WEB lors de l'authentification (mode basic,cf. RFC 2617). Cette variable fait partie des spécifications CGI décrites dans la RFC 3875.

Le langage utilisé pour le développement des `CGI` est `Perl`, complété par les packages `XML : :Simple` et `Image : :Magick` et leurs dépendances. Ce langage facilite la création de pages WEB grâce à son package `CGI` intégré.

La phase d'apprentissage/recherche de solution est confiée à `Scilab`. Ce logiciel a été utilisé au début du stage afin de visualiser l'évolution des contraintes les unes vis à vis des autres au cours de la recherche de solution, ce que ne permet pas aisément `C++` ou un autre langage compilé sans développements supplémentaires. Cet outil s'est révélé efficace après quelques optimisations et a été conservé afin de pouvoir utiliser sa bibliothèque d'algorithmes pour l'apprentissage des solutions, et notamment la programmation linéaire. Les difficultés liées à l'emploi de ce logiciel pour la phase de calcul sont :

1. La programmation spécifique à `Scilab` qui préfère les produits matriciels aux boucles classiques.
2. La mise en forme des données en entrée et en sortie afin de "communiquer" avec l'interface, et afin de satisfaire aux exigences de la programmation par produits matriciels.

Il y a, de plus, des traitements intermédiaires chargés de convertir les données d'un format à l'autre. Le format `XML` est utilisé pour stocker l'essemble des données du logiciel ce qui a permis un gain de temps en développement grâce à l'utilisation du package `XML : :Simple`. Une première conversion est nécessaire au commencement, afin de traduire les données relatives au système (bâtiments ouverts) et aux équipes (actives). Par la suite ces données ne doivent plus évoluer au cours de la session car sinon l'algorithme d'apprentissage pourrait se référer à des bâtiments fermés ou des équipes inactives. Ensuite, avant de lancer l'algorithme d'apprentissage et la recherche de solution, il est nécessaire de transformer les fichiers de préférences nouvelles et anciennes au format `scilab`. Enfin, aprés la recherche de solution, `Scilab` doit générer un fichier `XML` des solutions obtenues.



Avec étapes intermédiaires

5.3 La recherche adaptative en `Scilab`.

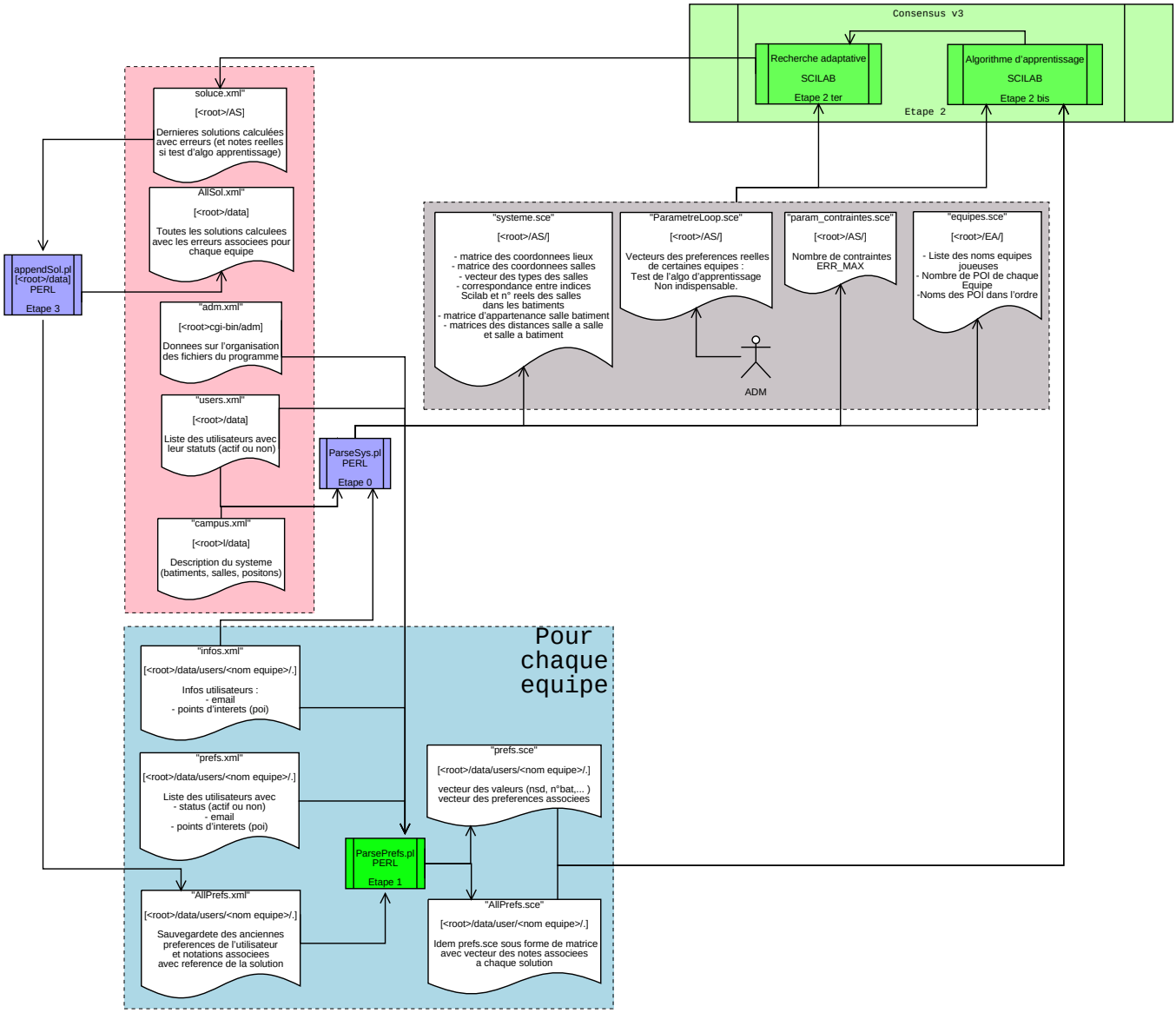
La recherche de solution implémentée permet maintenant de spécifier plusieurs contraintes de même type :

Besoins en salles : La différentiation des types de salles demandées est prise en compte. Un utilisateur peut demander un type particulier de salle. Le nombre de types n'est pas limité, et fait parti des données du système.

Distance à un lieu, à équipe : La notion de distance à un bâtiment est entendue à la distance à un lieu donné. De plus l'utilisateur peut maintenant dire s'il veut être proche ou loin d'un lieu ou d'une équipe. L'administreur fixe le nombre de contraintes de chaque type que les utilisateurs peuvent poser.

Dispersion :
Sans changement.

Le graphique ci-dessous montre les flux de données chargés d'alimenter les algorithmes d'apprentissage et de recherche de solution, et les traitements sur ces données (rectangles à double bordures droite gauche). Ce graphique traite de la partie calcul de consensus (phase 2). On peut y voir les scripts `Perl` de traduction de données `XML` en `Scilab`. La traduction des données `Scilab` en `XML` est réalisé par une fonction `Scilab` qui n'est pas représentée ici. Les parties actives sont en haut à droite de ce graphique. Elles sont détaillées dans le graphique d'après.

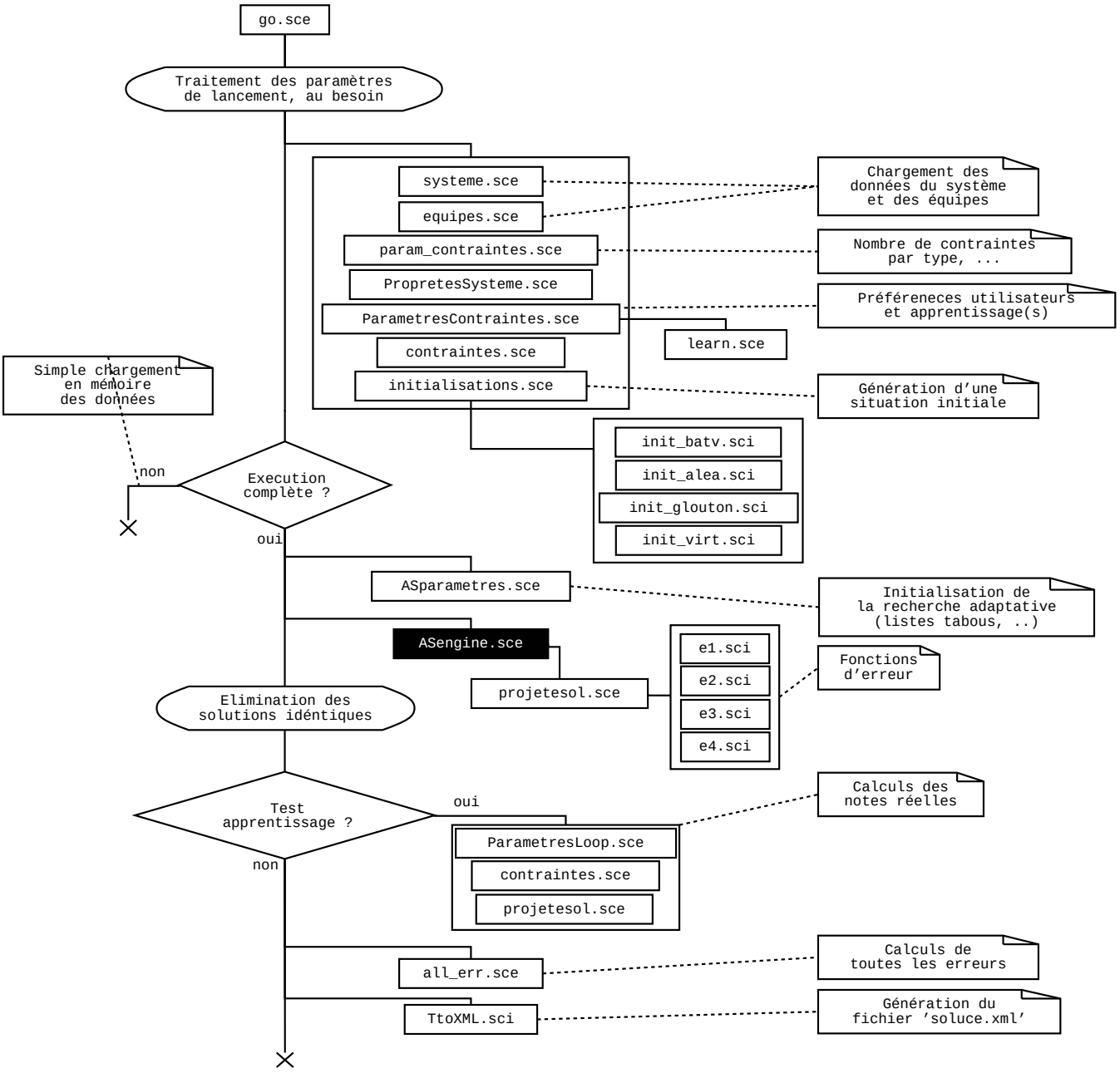


Fichiers de configuration, scripts de changements de formats, et modules de calcul.

On remarque sur ce graphique l'ensemble des fichiers de données nécessaires. Seuls ceux au format `XML` sont indispensables, les autres étant générés par les scripts `Perl`. L'exécution du script `ParseSys.pl` n'est cependant nécessaire qu'en début de session, ou en cas d'ajout

de points d'intérêts par les utilisateurs.

Le graphique suivant montre l'organisation de la partie apprentissage / recherche de solutions. L'ensemble se lance grâce au script `go.sce`. Ce script peut être lancé en ligne de commande avec des arguments de façon à être exécuté sans l'interface graphique de `Scilab` et sans boite de dialogue (cf. fichier `README`). A ce niveau, les fichiers de données sont supposés être à jour.


go.sce
Traitement des paramètres de lancement, au besoin
systeme.sce
equipes.sce
param_contraintes.sce
PropretesSysteme.sce
ParametresContraintes.sce
contraintes.sce
initialisations.sce
learn.sce
Chargement des données du système et des équipes
Nombre de contraintes par type, ...
Préféreneces utilisateurs et apprentissage(s)
Génération d'une situation initiale
Simple chargement en mémoire des données
non
Execution complète ?
oui
init_batv.sci
init_alea.sci
init_glouton.sci
init_virt.sci
ASparametres.sce
Initialisation de la recherche adaptative (listes tabous, ..)
ASengine.sce
projetesol.sce
e1.sci
e2.sci
e3.sci
e4.sci
Fonctions d'erreur
Elimination des solutions idéntiques
Test apprentissage ?
oui
non
ParametresLoop.sce
contraintes.sce
projetesol.sce
Calculs des notes réelles
all_err.sce
TtoXML.sci
Calculs de toutes les erreurs
Génération du fichier 'soluce.xml'

Etapes de la partie calculatoire.

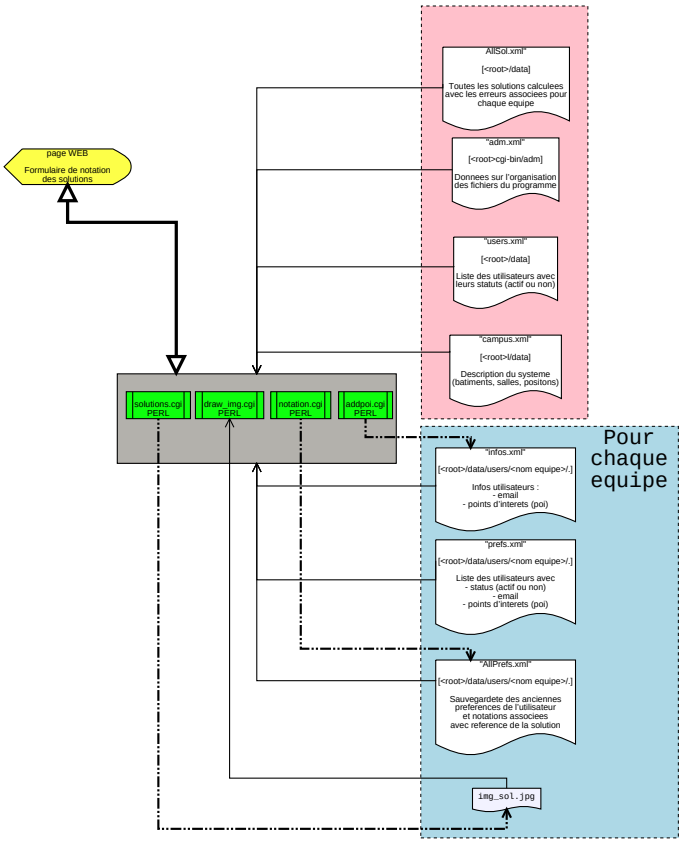
5.4 L'interface utilisateur.

Les utilisateurs accèdent à l'interface en tapant l'adresse du serveur WEB dans leur navigateur. L'accès est sécurisé par un mot de passe. Dans la version du logiciel décrite ici il n'est pas possible de changer ce mot de passe via l'interface. Cela est tout de même possible directement sur le serveur. Le nom de l'utilisateur est celui de son équipe de travail.

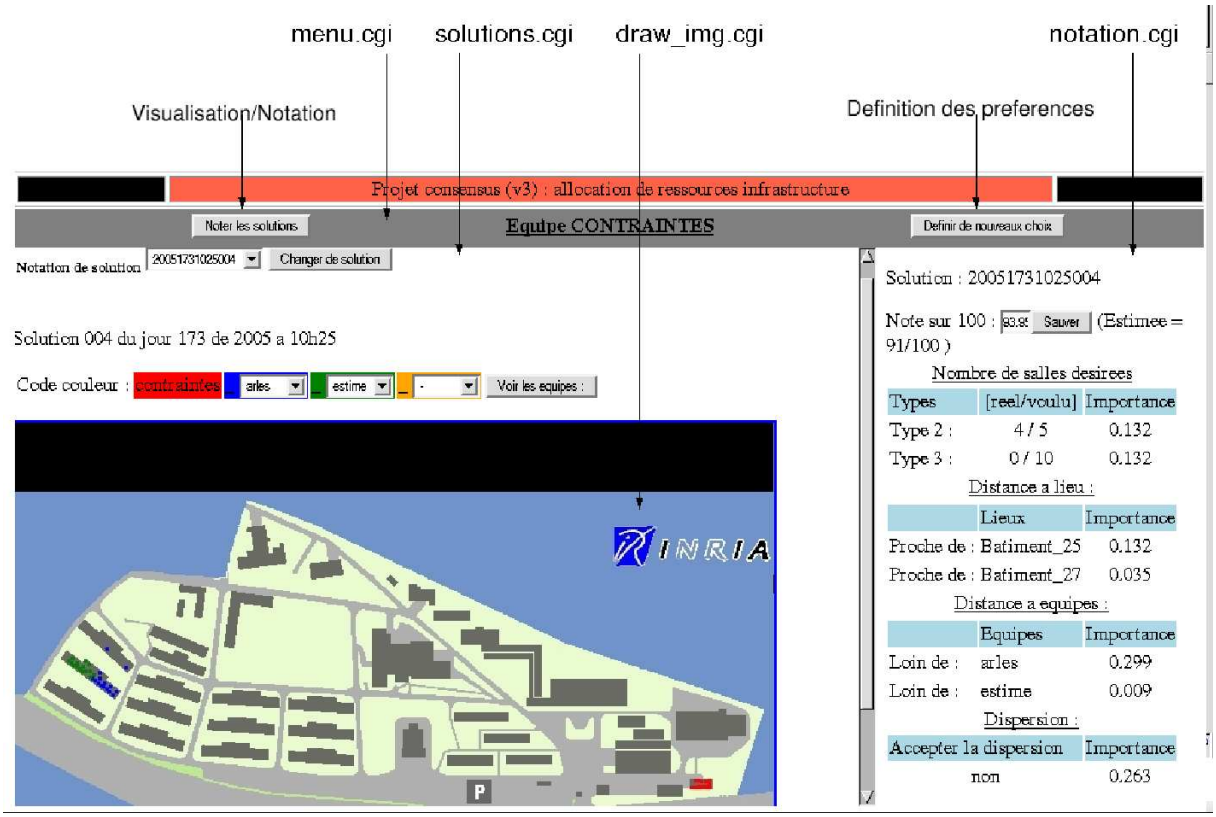
L'interface WEB repose sur deux groupes de script `CGI` écrits en `Perl`. L'un est chargé de présenter les préférences de l'utilisateur et de permettre leurs modifications, l'autre permet la visualisation et la notation des solutions. Ces scripts s'appuient sur les mêmes fichiers de données `XML` que la partie calcul en `scilab`.

5.4.1 Visualisation, notation

L'interface présentant les solutions et permettant la notation est la suivante :



Visualisation, notation.



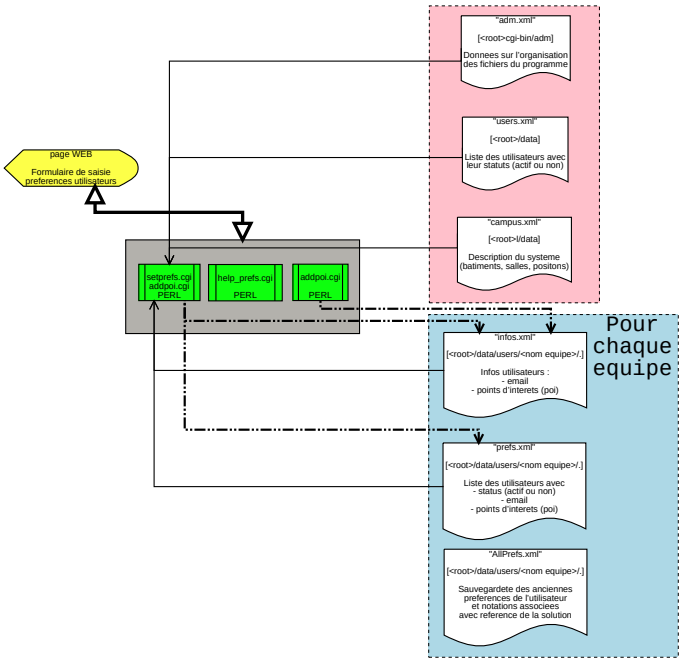
Interface pour la visualisation et la notation de solutions.

La partie droite de l'interface donne les préférences utilisées pour le calcul de la solution présentée. Elle présente la note estimée de façon à ce que l'utilisateur puisse noter en rapport.

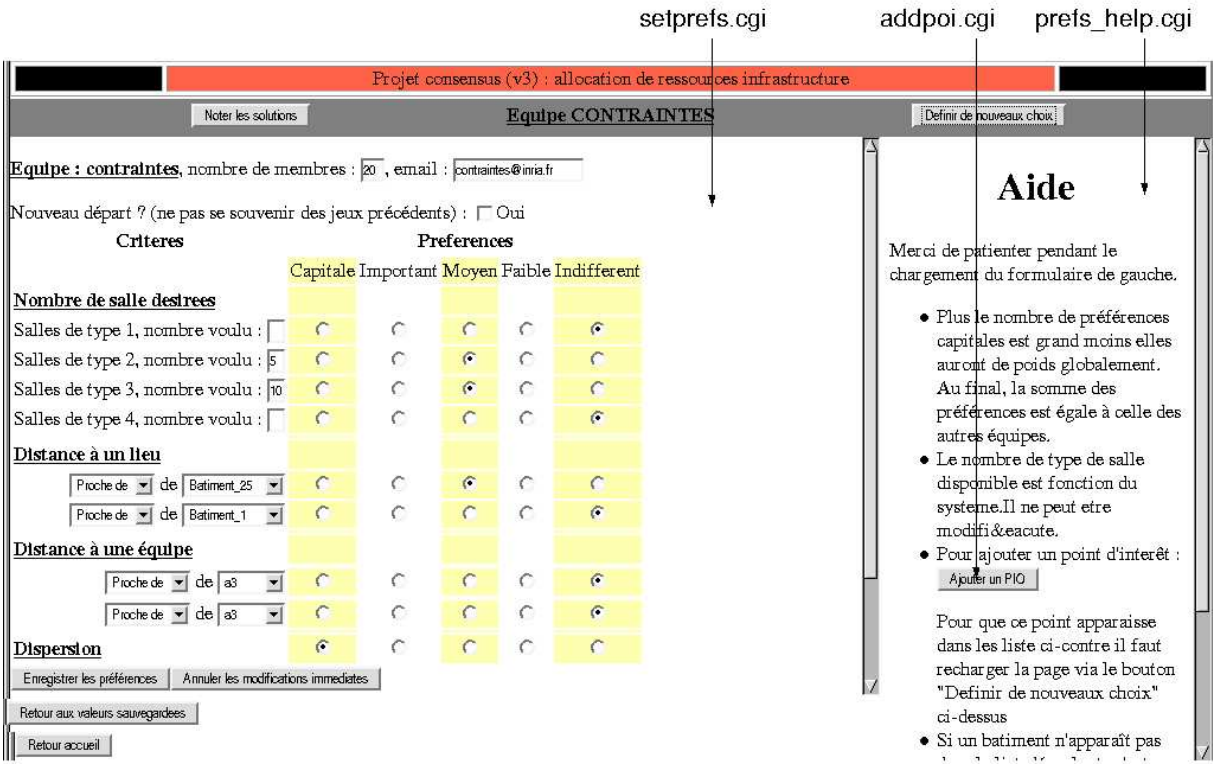
La partie gauche permet de sélectionner la solution à voir ainsi que les positions de quelques équipes. En cliquant sur un bâtiment on obtient l'occupation des salles de ce bâtiment. Au moment du développement certains plans de bâtiment manquaient. Dans ce cas l'interface n'affiche aucune image et il est nécessaire de relancer l'affichage des solutions par le bouton du menu. Un nouveau clic sur l'image du bâtiment permet de revenir à l'image du campus. L'image affichée est calculée en fonction des demandes de l'utilisateur grâce au module `Image : :Magick`. Les images de bases sont stockées dans un répertoire commun. Les images calculées sont stockées dans les répertoires propres à chaque équipe.

Par défaut, c'est la dernière solution calculée qui est présentée. Le numéro d'une solution est composé de la date du calcul (année, quantième,heures,minutes), et du numéro d'ordre de la solution.

5.4.2 Expression des préférences



Expression des préférences.



Interface pour l'expression des préférences.

La partie gauche permet aux utilisateurs de renseigner leurs préférences. Seul le nombre de contraintes fixé par l'administrateur apparaît. Les poids des contraintes se fixe sur une échelle à 5 niveaux. La partie droite est une aide. Il est possible d'ajouter un point d'intérêt via le bouton adéquat figurant dans cette partie. Toutefois, cette fonction peut ne pas être opérationnelle dans la version finale. Par défaut, le dernier jeu de préférences fixé par l'utilisateur est affiché. Si aucun jeu de préférences n'a été défini, toutes les contraintes sont considérées de poids nul.

5.5 Déroulement typique d'une session.

Voici dans l'ordre les actions à réaliser par un administrateur pour assurer le déroulement d'une session :

Initialisation :

1. Définir les équipes joueuses, les bâtiments ouverts, les salles ouvertes (fichiers `campus.xml` et `equipes.xml`).

2. Exécuter le script `filluser.pl` du répertoire `bin/` afin de créer ou de recréer les répertoires utilisateurs et les fichiers utilisateurs.
3. Exécuter le script `ParseSys.pl` afin de générer les fichiers scilab décrivant le système.

Nota : le format `XML` utilisé pour les fichiers de configuration doit faciliter l'écriture d'une interface pour l'administrateur.

Jeux :

1. Permettre aux joueurs d'exprimer leurs préférences, et de noter les solutions si elles existent par l'activation du serveur WEB.
2. Lorsque la première phase est terminée, lancer le script `ParsePrefs.pl` afin de convertir les préférences au format `scilab`.
3. Lancer la deuxième phase grâce au script scilab `go.sce` en ligne de commande avec arguements (cf `README`) ou en mode fenêtré pour voir les évolutions de la fonction objectif.
4. Lorsque les calculs sont terminés, lancer le script `appendsol.pl` afin d'ajouter les solutions trouvées à celles calculées précédemment, et d'ajouter à chaque équipe les jeux de préférences appris pour le calcul des dernières solutions.

Il n'y a pas d'opération finale à réaliser. Le fichier `AllSol.xml` contient l'ensemble des solutions calculées dans l'ordre. On peut concevoir une transformation de ce fichier en un formulaire d'affectation plus contractuel en fin de session.

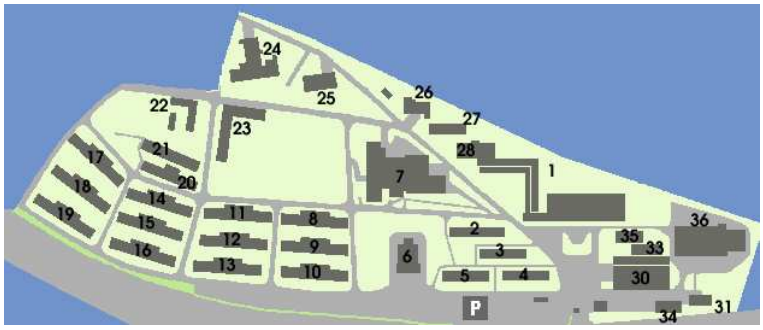
5.6 Paramétrage

Le fichier `adm.xml` permet de définir les paramètres généraux du logiciel (les noms des fichiers et des répertoires), ou les paramètres de la recherche adaptative (le nombre de contraintes par type), ou l'erreur maximale. Ce fichier doit être à la racine du projet. Il est utilisé par tous les scripts `Perl`.

6 Evaluation

Nous étudions ici le cas d'une session mettant en concurrence dix équipes sur le campus complet de l'INRIA.

Les demandes des utilisateurs en terme de salles et de dispersion sont les mêmes :
- 5 salles de type 2, coefficient = 0.2
- 10 salles de type 3, coefficient = 0.2
- non dispersé, coefficient = 0.4



Numéros des bâtiments

Individuellement chaque équipe pose soit une contrainte de type 2, soit une contrainte de type 3. Le poids de cette contrainte est de 0.4 pour chaque équipe. Soit, par numéro d'équipe (les noms des équipes sont données à titre d'exemple, leurs préférences sont purement illustratives)

Numéro	nom	bâtiment	équipe
1	a3	proche 17	
2	algo	proche 10	
3	arles		proche a3
4	atoll	proche 14	
5	cristal	proche 2	
6	contraintes	proche 25	(loin a3)
7	daf		proche atoll
8	dirrocq	proche 10	
9	eiffel	loin 17	
10	estime		loin eiffel

Cette instance inclut un accès concurrent au bâtiment 10 (équipes 2 et 8). Dans notre test, ce bâtiment compte 18 salles alors que 30 sont demandées au total par les deux équipes. Pour l'équipe 6, la préférence entre parenthèse est celle cachée, par opposition à celle déclarée. C'est la seule équipe pour laquelle l'algorithme d'apprentissage intervient.

Ce test correspond à celui utilisé pour vérifier l'algorithme d'apprentissage (chapitre 4). Le présent chapitre rend compte de l'exécution d'une session de deux fois quatre jeux. Les critères d'arrêt de la recherche adaptative sont : 3000 itérations ou 3 restarts. (voir traces en annexe C).

6.1 Performances

Jeux	Durée	Restart (itérations)	Dernière itération
1	02H03	557, 1043, 1529	1971
2	02H55	496, 1205, 1664	2255
3	02H13	520, 999, 1440	1955
4	02H06	521, 997, 1491	2034
5	02H36	611, 1184, 1750	2233
6	02H27	502, 996, 1437	1921
7	02H06	486, 980, 1421	1904
8	02H21	486, 993, 1434	2005

A noter que les délais indiqués ci-dessus sont des maxima. La machine utilisée pour les calcul (pentium 4, 3GHz) est partagée avec un autre utilisateur, et l'exécution du calcul n'est pas prioritaire.

Chaque jeu génére 4 solutions notées par équipe 6.

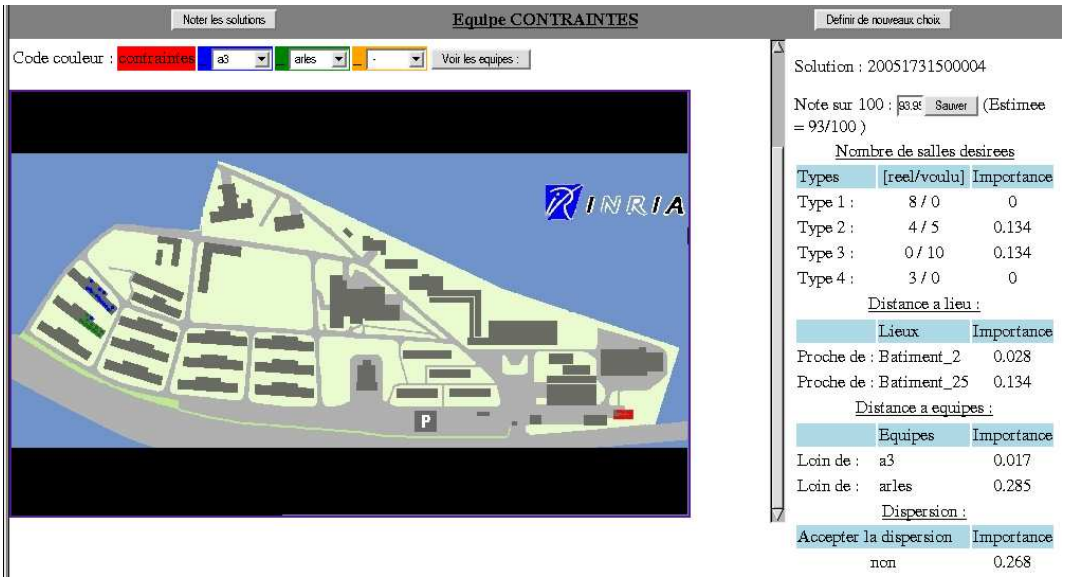
La durée moyenne d'exécution se situe environ à 02H15. Le nombre d'itérations total moyen est de 2035. Soit une moyenne de 15 itérations par minute pour cette instance.

6.2 Résultats

Les répartitions données ci-dessous sont celles de la solution 4 du jeu 7.

Equipes 1, 3, et 6

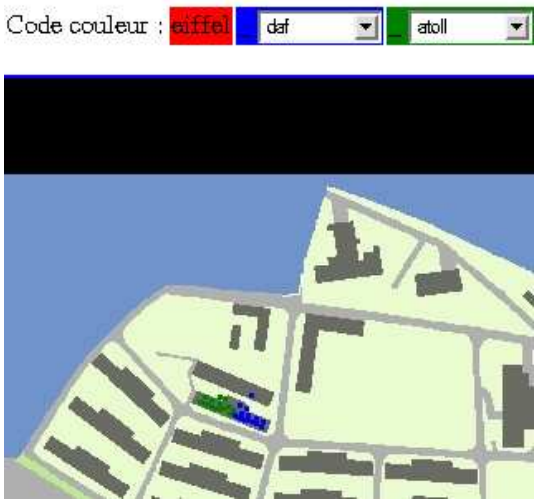
Le graphique du dessous représente les répartitions des équipes a3(1), arles(3) et contraintes(6). On vérifie que l'équipe 6 est placée en conformité avec ses préférences cachées, c'est à dire loin de l'équipe 1. L'algorithme d'apprentissage oriente donc correctement les préférences de l'équipe 1. Par ailleurs, l'équipe 1 est correctement placée dans le bâtiment qu'elle désirait, et l'équipe 3 est correctement placée à proximité de l'équipe 1.



Equipes 4 et 7

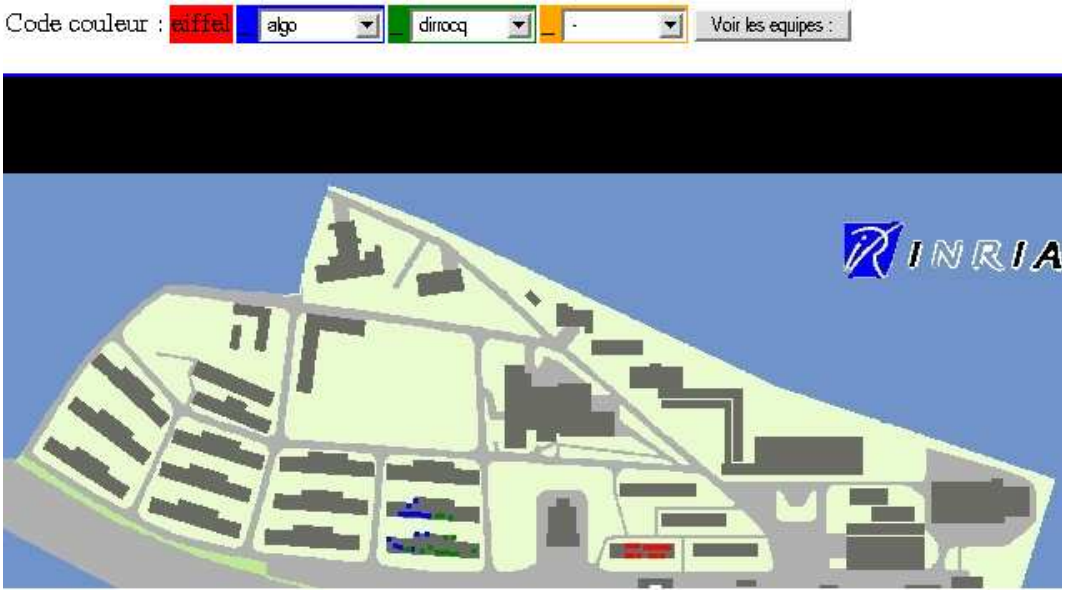
Le graphique ci-dessous montre que l'équipe atoll(4), qui visait le bâtiment 14, n'a pas pu s'y loger. Ce bâtiment, bien que déclaré ouvert, n'a pas de salle ouverte. Elle a été placée dans le bâtiment 20 juste au nord, ce qui est satisfaisant. Par ailleur, on vérifie que l'équipe daf(7) a correctement été positionnée prés de l'équipe 4.

L'équipe eiffel(9) est traitée plus bas.



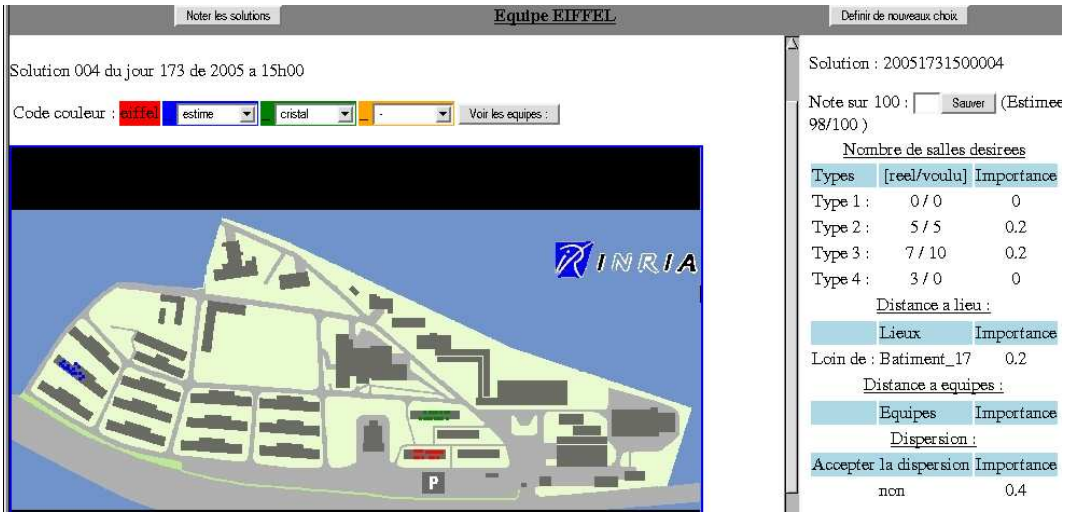
Equipes 9, 2, et 8

Le graphique ci-dessous met en évidence l'accès concurrent au bâtiment 10 par les équipes algo(2) et dirrocq(8). On voit que ces deux équipes sont assez bien satisfaites. Par ailleur, l'équipe eiffel(9) a correctement été positionnée loin du bâtiment 17.



Equipes 6 et 10

L'équipe estime(10) est correctement positionnée loin de l'équipe eiffel(9), et l'équipe cristal(6) occupe le bâtiment demandé.



De façon générale, les équipes sont regroupées comme le laissait supposer le coefficient relativement fort sur cette contrainte. Ces regroupements forcent le viol des contraintes NSD et on peut vérifier que ce viol est d'autant plus grand que les coefficients sur les contraintes de type 1 sont faibles. Ainsi, si pour eiffel le type de salle respecté pour 12 salles sur 15, pour contraintes il ne l'est plus que pour 4 salles sur 15. Concernant la répartition des équipes algo et dirrocq qui ont demandé le même bâtiment, on constate également un bon respect du type de salle. Ce résultat est bon et tend à valider le calcul de normalisation des contraintes de type 1 vis à vis des autres.

6.3 Mémoire

Les quantités de mémoires nécéssaires sont importantes pour scilab et les scripts perl. Pour `Scilab`, le besoin en mémoire s'explique par la nécéssité de stocker des matrices qui sont d'ordres supérieurs à 700x700 dans notre exemple. Ces matrices caractérisent le système (distances entre salles, appartenance à un bâtiment). La recherche adaptative en elle-même n'est pas très couteuse en mémoire. Les listes tabous sont de tailles limitées (N pour la liste tabou sur les équipes, $\max_{n\in\{1,\dots,N\}} P^n$ pour la liste sur les salles). Les scripts `Perl` ont quant à eux besoins de mémoire pour traiter les fichiers `XML` qui atteignent 2 Mo dans le cas présent. La rapidité du parseur `XML` doit être importante car plus les fichiers sont volumineux plus l'interface est lente.

7 Conclusion

L'outil développé offre maintenant la possibilité de spécifier plusieurs contraintes d'un même type, et traite le problème d'allocation de bureaux en vraie grandeur. L'étude théorique et pratique des fonctions d'erreur garantit pour chaque contrainte que sa violation est valuée dans un intervalle normalisé, avec une distribution relativement homogène. L'apprentissage des préférences utilisateur est maintenant résolu de façon exacte par un programme linéaire.

Une séparation nette entre la partie interface WEB et la partie recherche de solution et réalisée de fait par l'utilisation de `Scilab`. Le format des fichiers de données (`XML`) permet, dans une certaine mesure, de faire évoluer tout ou partie de l'outil de façon indépendante.

L'outil développé génère des solutions correctes. Cependant il manque de rapidité ce qui impose le travail par phases successives. Les optimisations envisageables sont de plusieurs ordres :

- Adoption de grands voisinages, considérant des permutations par blocs plutôt qu'unitaires, afin d'accélérer la descente vers les extrema locaux.
- Réduction du domaine de recherche des permutations envisageables, prise en compte de la géométrie du système.
- Optimisation des scripts`Scilab`, ou transformation en fonctions pré-compilées.
- Utilisation d'un parseur `XML` plus rapide pour le chargement des données dans l'interface graphique.
- Exploitation des fonctions de calcul parallèle de `Scilab`, certaines fonctions d'erreur pouvant être évaluées conjointement.
- Scission du campus en parties tenant compte des intérêts d'un groupe d'équipes pour une plus petite partie du campus, de façon à restreindre la recherche à cette partie et à ce groupe d'équipes.

Une autre amélioration mineure possible concerne la distance à un bâtiment. Plutôt que de considérer les distances par rapport à un point central, il serait plus judicieux de considérer la distance minimale par rapport aux issues du bâtiment, ce lorsqu'une équipe désire la proximité d'un bâtiment et que toutes ses salles ne sont pas dedans. On pourrait éviter de la sorte certaines affectations correctes du point de vue de la dispersion, mais éloignées des issues.

Par ailleurs, la possibilité de partir d'une solution antérieure ou d'une situation existante n'est pas implémentée.

Enfin, il faut noter que l'outil n'a pas été testé dans toutes les configurations possibles et à pleine charge (avec les 20 équipes de l'INRIA). Ceci permettrait en particulier de tester dans ce contexte la gestion des salles virtuelles, étudiée pour le moment sur de petites instances.

A Glossaire

- Campus : ensemble clos de bâtiments.
- Contrainte hétérogène : contraintes dont les fonctions d'erreur ont des valeurs de grandeurs différentes (ex : nombre de salles désirées, et distance à un bâtiment)
- ERR_MAX : valeur de l'erreur maximale pour toutes les fonctions d'erreur. Egalement note maximale attribuable à une solution.
- Fonction d'erreur : la recherche adaptative lie une fonction d'erreur à chaque contrainte du problème. Cette fonction est nulle si la contrainte est vérifiée, et croît à mesure que la contrainte est violée.
- Fonction objectif : fonction dont on cherche à trouver un optimum. Ici on cherche à minimiser l'insatisfaction de l'équipe la plus insatisfaite.
- Initialisation : détermination d'une situation de départ pour un algorithme d'optimisation.
- Liste tabou : mémoire utilisée pour interdire temporairement l'exploration de certaines variables.
- Note : évaluation par un utilisateur d'une situation calculée. D'autant plus grande que la situation satisfait l'utilisateur (ERR_MAX−insatisfaction).
- NSD : abréviation pour "nombre de salles désirées".
- Préférence, paramètre de contrainte : valeur liée à un type de contrainte. Peut être un numéro de bâtiment, d'équipe.
- Programmation linéaire : algorithme d'optimisation continue exact, à contraintes linéaires.
- Recherche adaptative : méthode d'optimisation combinatoire, à recherche locale de solution.
- Recherche locale : caractéristique de méthodes d'optimisation recherchant des solutions intermédiaires de coûts strictement inférieurs aux coûts précédents.
- `Scilab` : logiciel gratuit pour le calcul mathématique non formel.
- Système : ici défini comme l'ensemble des salles, des bâtiments et des équipes.
- Variables : ici chaque salle est une variable de valeur le numéro de l'équipe utilisatrice.
- Parseur : partie d'un programme chargé de lire un fichier et de renseigner des variables du programme avec les données contenues dans le fichier.
- `Perl` : langage de script adapté au traitement des fichiers textes et à la génération de pages `WEB`.
- Scripts `CGI` : programmes générant dynamiquement des pages `WEB` et les transmettant au serveur `WEB` pour diffusion.
- Serveur `WEB` : programme répondant aux demandes de connection des navigateurs distants en envoyant des pages HTML.
- `XML` : format standard d'enregistrement des données sous forme arborescente.

B La recherche adaptative en `Scilab`, version 'moyenne minimum'

Dans sa forme originale, le logiciel `CONSENSUS` initie l'algorithme de recherche adaptative par une solution particulière : les équipes sont entièrement logées dans des salles virtuelles. Pour cela, le logiciel ajoute autant de salles virtuelles que de salles demandées. Cette fonctionnalité est reprise dans la version `Scilab` de `Consensus`, développée afin de valider les contraintes et de mieux tracer l'exécution de l'algorithme.

Le présent paragraphe donne les résultats obtenus lors des recherches de solutions minimisant la moyenne des erreurs sur les équipes. Le but est de savoir dans quelle mesure une contrainte prend le pas sur l'autre. Dans les exemples suivants, les équipes posent les mêmes coefficients sur toutes leurs contraintes.

Le premier exemple sert à confirmer le logiciel et faire les premières constatations. Le second visualise la prise en compte des préférences utilisateur.

La version `Scilab` de la recherche adaptative a été validée sur une petite instance qui n'est pas présentée ici. La mise à l'échelle est réalisée avec le campus complet de l'`INRIA`. L'erreur maximum est fixée à 100.

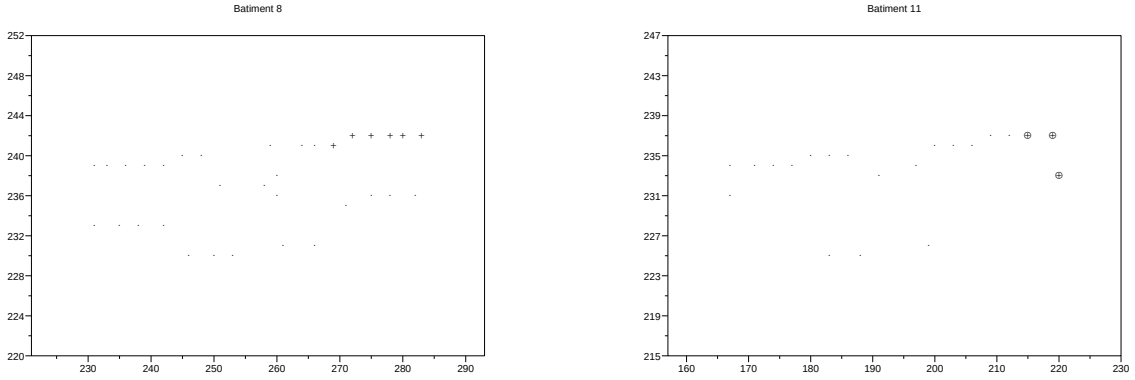
B.1 Jeu 1

Pour ce jeu on considère 3 équipes :
- La première demande 15 salles (coef $\frac{1}{2}$) et veut le bâtiment 8 (coef $\frac{1}{2}$).
- La deuxième demande 10 salles (coef $\frac{1}{3}$), veut le bâtiment 8 (coef $\frac{1}{3}$), et veut être groupée (coef $\frac{1}{3}$).
- La troisième demande 8 salles (coef $\frac{1}{2}$) et veut le bâtiment 11 (coef $\frac{1}{2}$).

La situation initiale fixe toutes les équipes dans des salles virtuelles. L'erreur sur la première contrainte est donc maximale pour toutes les équipes et vaut 50 pour les équipes 1 et 3, et $\frac{100}{3}$ pour l'équipe 2. Les premières itérations de l'algorithme montrent que les équipes 1 et 3 sont traitées en premier. A chaque fois que l'équipe 1 gagne une salle réelle, l'erreur sur sa première contrainte diminue et c'est alors l'équipe 3 qui est servie. La première salle n'est attribuée à l'équipe 2 qu'à l'itération 10 alors que l'équipe 1 dispose de 6 salles réelles et que l'équipe 3 dispose de 3 salles réelles. A noter que l'algorithme satisfait les contraintes 2 des équipes 1 et 3 liées au choix du bâtiment en placant directement ces équipes dans les bâtiments demandés, ces permutations étant de moindres coûts.

A noter également que l'erreur liée au nombre de salles attribuées décroît moins vite pour l'équipe 1 que pour l'équipe 3 du fait que l'équipe 1 demande plus de salles. L'équipe 1

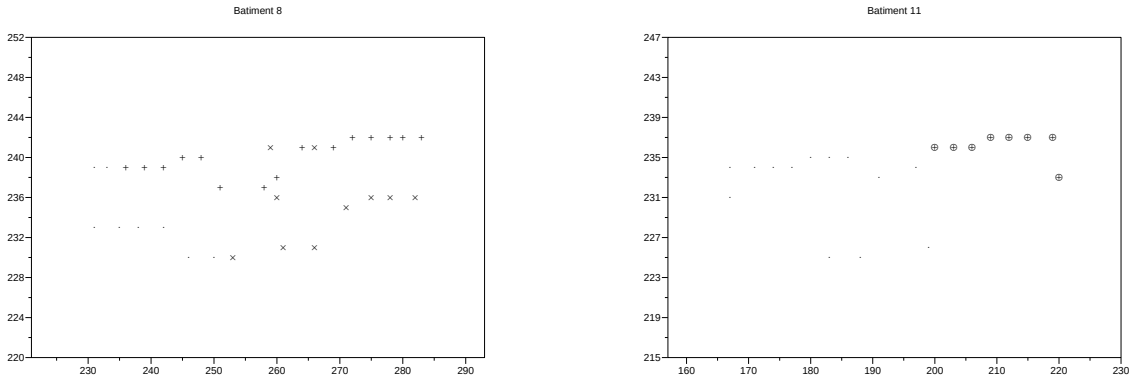
est donc plus souvent servie que l'équipe 3. Ceci est conforme à ce qui est souhaité. L'équipe 3 pourrait obtenir des salles au même rythme en augmentant son coefficient de préférence.



Situation des batiments 8 et 11 à l'itération 9 (une marque par salle).
Marques = (.) : salle vide, (+) : équipe 1, (×) : équipe 2, ($\oplus$) : équipe 3.

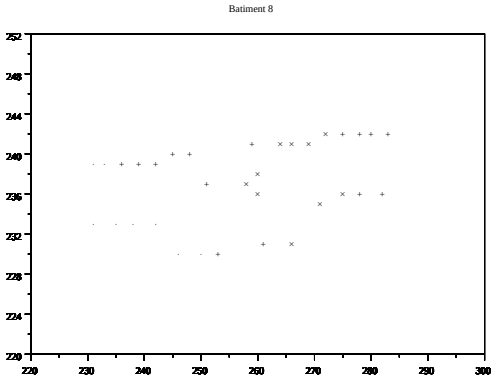
L'algorithme poursuit ses affectations en donnant une salle à l'équipe 2 dès que l'erreur liée au nombre de salles attribuées est plus élevée pour l'équipe 2 que pour les autres équipes. Les besoins exprimés par les équipes étant réalisables on arrive à une situation où seule la contrainte de dispersion posée par l'équipe 2 n'est pas nulle. Cette contrainte ne peut pas être prise en compte tant qu'il reste des équipes en salles virtuelles, car l'erreur commise s'il manque une salle à une équipe est supérieure à l'erreur maximum atteignable par l'erreur 4 de dispersion posée par l'équipe 2. S'il manque une salle, la fonction d'erreur 1 vaut respectivement $\frac{10}{3}$, $\frac{10}{3}$ et $\frac{25}{4}$. La fonction d'erreur 4 posée par l'équipe 2 a un maximum du fait de la possibilité d'affecter toutes les salles demandées dans les bâtiments demandés. Ce maximum est le quotient de la distance maximum séparant deux salles du bâtiment 8 par la distance maximum séparant deux salles du campus, coefficienté par le poid donné à la contrainte et par la normalisation. Ce maximum est ici de l'ordre de 3.048.

La situation à l'itération 33 est donc telle qu'il ne reste plus que la contrainte 4 de l'équipe 2 à minimiser.



Situation des bâtiments 8 et 11 à l'itération 33.

Enfin l'algorithme converge vers la solution de moindre coût suivante :



Situation finale du batiments 8 à l'itération 58, bâtiment 11 inchangé.

A noter qu'il est possible d'obtenir des situations où le coût n'est visiblement pas minimum et où il n'y a pas d'échange de la variable de coût maximum strictement améliorant. On sort de ces situations par l'utilisation d'une liste tabou.

En conclusion de ce premier test on voit que les contraintes se comportent conformement à ce qu'on peut attendre, ainsi que la liste tabou.

B.2 Jeu 2

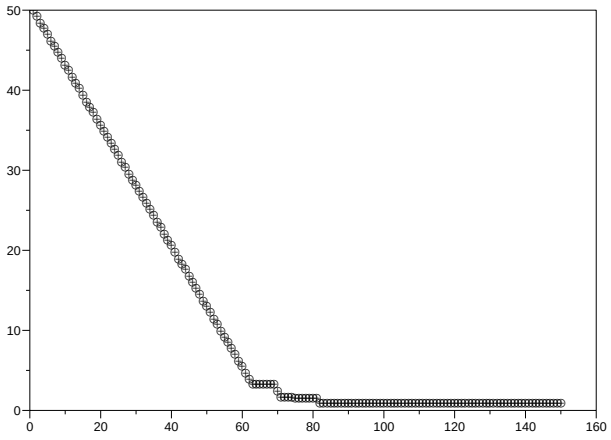
Pour ce jeu on considère les mêmes équipes mais sur un campus réduit afin de limiter les délais de calculs. Afin que ce jeu soit comparable avec le précédent, seuls les bâtiments les plus éloignés sont enlevés (n°24 et au-delà). Le problème est alors de dimension 574 au lieu de 730 (nombre de salles reélles).

- La première équipe demande 20 salles (coef $\frac{1}{2}$) et veut le bâtiment 8 (coef $\frac{1}{2}$).
- La deuxième équipe demande 20 salles (coef $\frac{1}{2}$), veut le bâtiment 8 (coef $\frac{1}{10}$), et veut être groupée (coef $\frac{2}{5}$).
- La troisième équipe demande 25 salles (coef $\frac{1}{2}$) et veut le bâtiment 11 (coef $\frac{1}{2}$).

Ce test est conçu de telle sorte que les bâtiments visés soient saturés, et afin de mettre en évidence la préférence de l'équipe 2 sur son regroupement plutôt que sur sa position.

La situation initiale est la même que précédement : les équipes partent des salles virtuelles.

L'évolution des coûts est donnée par le graphique suivant qui montre que ce coût décroit linéairement jusqu'à l'itération 62. Comme pour le premier jeu cette phase correspond aux permutations des salles virtuelles vers les salles réelles. Cette fois les coefficients sur la contrainte 1 (nombre de salles demandées) sont les mêmes, et on constate que les affectations se font de la même façon qu'au jeu 1 sur toutes les équipes. Cependant cette affectation s'arrête alors qu'il reste une salle virtuelle afféctée à chaque équipe.

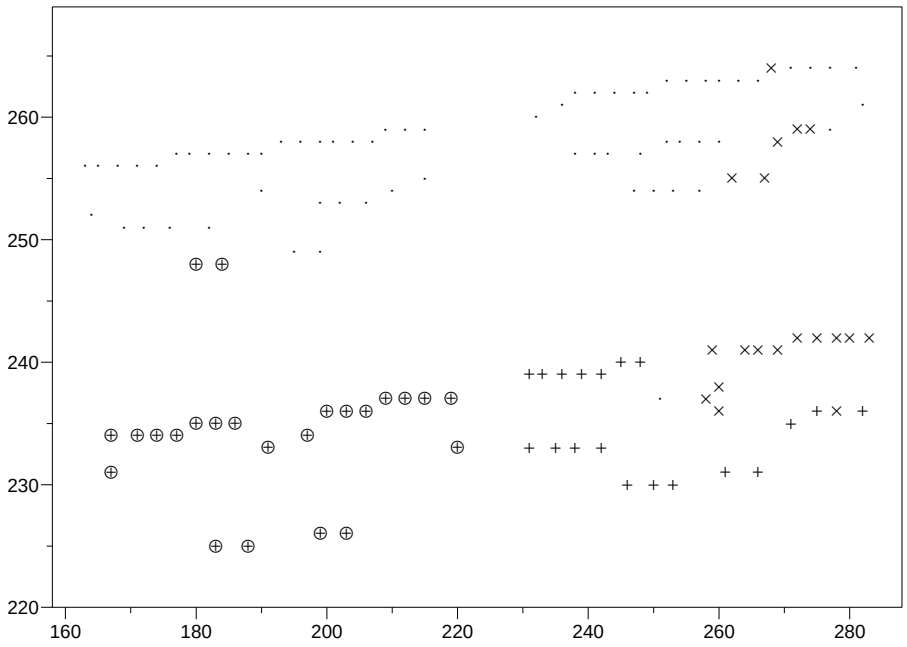


Puisque les demandes en salles sont supérieures aux capacités des bâtiments visés, ont constate que :

– l'équipe 1 occupe seulement le bâtiment 8,
– l'équipe 2 est répartie dans les bâtiments 8 et 9,
– l'équipe 3 est répartie dans les bâtiments 11 et 12,

Ces résultats sont cohérents avec ce qu'on pouvait attendre. Les bâtiments 9 et 12 sont les plus proches des bâtiments visés (respectivement 8 et 11). L'équipe 2 a posé une préférence cinq fois moins forte sur l'occupation du bâtiment 8 que l'équipe 1.

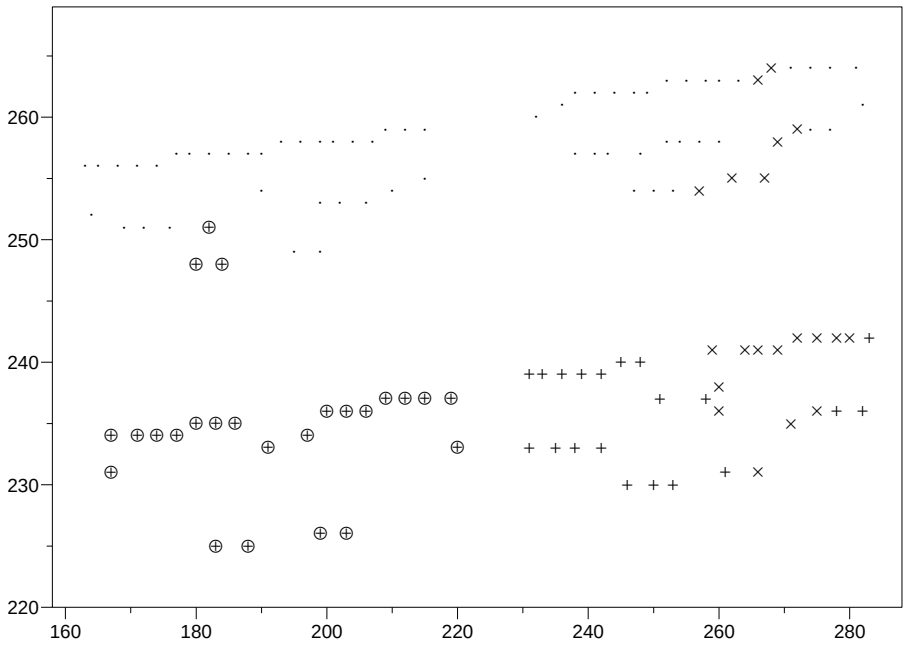
La situation à l'itération 62 est la suivante :



Situation des bâtiments 12, 9, 11 et 8 à l'itération 62 (dans le sens de lecture).
(.) : salle vide, (+) : équipe 1, (×) : équipe 2, (⊕) : équipe 3.

Les transferts des salles virtuelles vers les salles réelles s'arrête donc à l'itération 62. Cela s'explique car les valeurs des projections sur les salles du bâtiment 9 sont supérieures aux projections sur les salles virtuelles. L'affectation de l'équipe 2 dans le bâtiment 9 provoque le viol des deux contraintes liées au bâtiment désiré et à la dispersion.

Le plateau observé sur la courbe des coûts entre les itérations 62 et 69 correspond à une recherche d'échange des salles de l'équipe 2 dans le bâtiment 9. Il n'y a pas d'amélioration possible, la liste tabou remplit son rôle et permet de sortir de ce plateau. Les deux baisses du coût observées aux itérations 69 et 81 correspondent aux permutations des dernières salles virtuelles sur les salles réelles.
Au delà de l'itération 81, le système ne trouve pas de permutation améliorante. On obtient au final la situation suivante :



Situation des bâtiments 12, 9, 11 et 8 à l'itération 148 (dans le sens de lecture).
(.) : salle vide, (+) : équipe 1, (×) : équipe 2, (⊕) : équipe 3.

En pratique la répartition de l'équipe 3 dans le bâtiment 12 n'est pas optimale. Les trois salles seraient mieux positionnées près des issues du bâtiment à droite ou à gauche. Cependant, la solution trouvée est conforme à la modélisation qui ne tient pas compte des entrées des bâtiments.

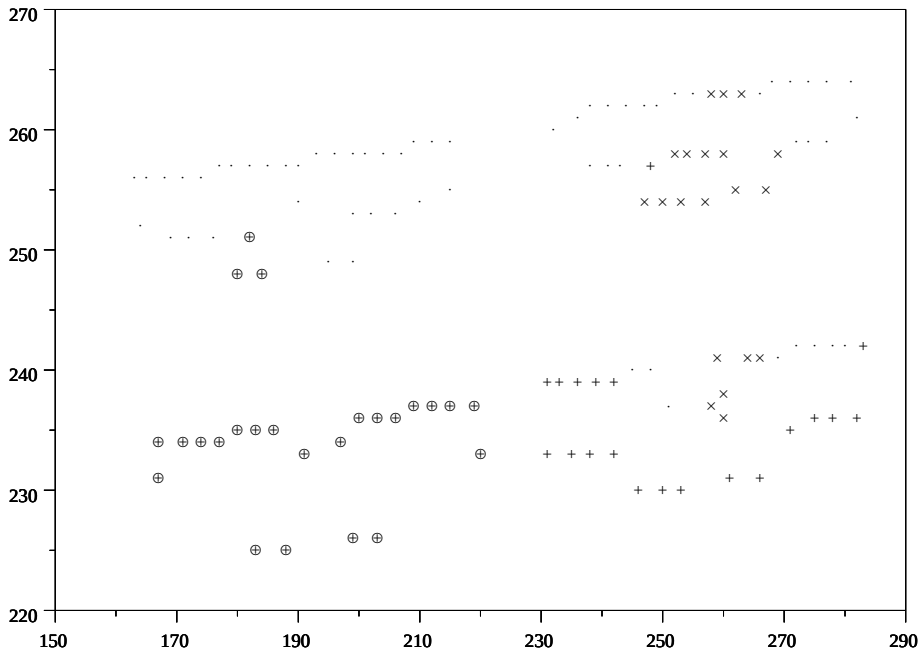
En conclusion ce jeu confirme la validité des fonctions d'erreur 1, 2 et 4 liées respectivement au nombre de salles désirées, au bâtiment visé, et à la dispersion. Le jeu suivant intègre la contrainte 3 liées à l'équipe visée.

B.3 Jeu 3

Le logiciel Scilab a été optimisé entre le test précédent et celui-ci, ce qui permet maintenant de travailler sur le campus complet de l'INRIA. Les résultats restent comparables car les solutions font intervenir les mêmes bâtiments.

Pour ce jeu :

- La première équipe demande 20 salles (coef $\frac{1}{2}$) et veut le bâtiment 8 (coef $\frac{1}{2}$).
- La deuxième équipe demande 20 salles (coef $\frac{1}{2}$), veut être proche de l'équipe 1 (coef $\frac{2}{5}$), et veut être groupée (coef $\frac{1}{10}$).
- La troisième équipe demande 25 salles (coef $\frac{1}{2}$) et veut le bâtiment 11 (coef $\frac{1}{2}$).



Situation des bâtiments 12, 9, 11 et 8 à l'itération 160 (dans le sens de lecture).
(.) : salle vide, (+) : équipe 1, (×) : équipe 2, (⊕) : équipe 3.

On peut vérifier ici le bon comportement de la contrainte distance à équipe.

C Etude du comportement des fonctions d'erreur

C.1 Cas fonction objectif 'Moyenne Minimum'

Il s'agit dans cette partie d'étudier le comportement relatif des contraintes lorsqu'elles sont mises en concurence.

La fonction objectif des précédents développements est conservée. On cherche des solutions qui minimisent la moyenne des erreurs des équipes.

On part de systèmes simples faisant jouer une ou deux équipes qui expriment chacune deux contraintes : une liée au nombre de salles désirées, et l'autre étant celle dont on veut étudier le comportement vis-à-vis de l'autre. De façon à pouvoir être comparées, les préférences sur les contraintes sont identiques et on les oublie dans un premier temps.

C.1.1 Confrontation (nombre de salles désirées) - (nombre de salles désirées)

On considère deux équipes ne posant chacune qu'une contrainte de type NSD (nombre de salles désirées). Supposons que $P^{n_1} \leq K^b$, $P^{n_2} \leq K^b$, et $(P^{n_1} + P^{n_2}) > K^b$. Dans ce cas l'erreur globale s'écrit :

$$E(T) = E_1^1(T) + E_1^2(T) = 1 - \frac{\phi(n_1,T)}{P^{n_1}} - \frac{\phi(n_2,T)}{P^{n_2}}$$

L'erreur est minimale si toutes les salles réelles sont occupées car cette fonction est linéaire strictement décroissante en $\phi(n_1,T)$ et en $\phi(n_2,T)$. Donc on a la relation $K = \phi(n_1,T) + \phi(n_2,T)$ D'où :

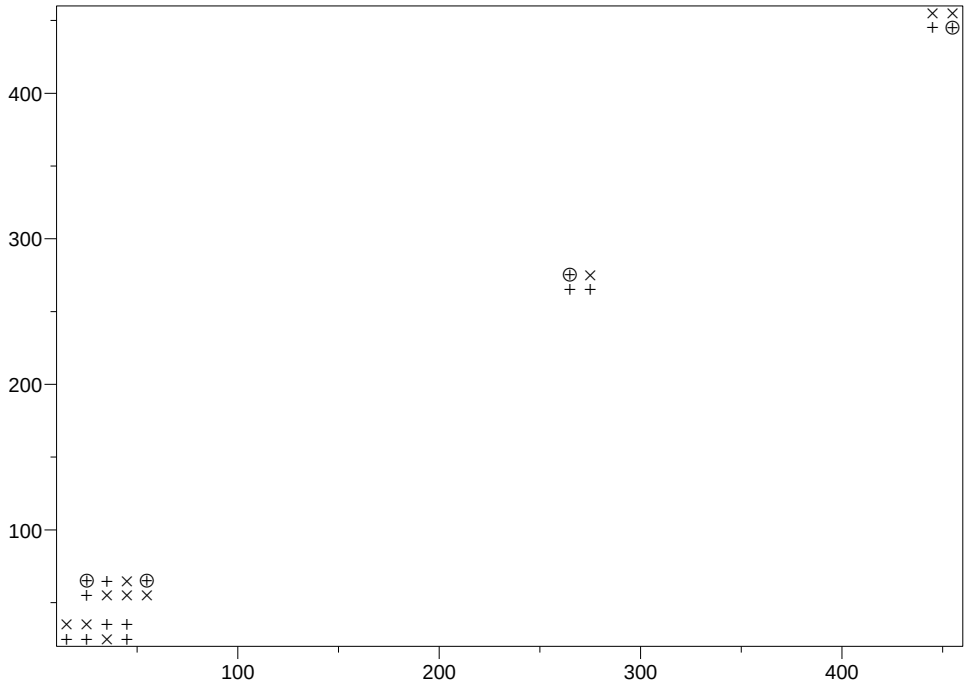
$$E(T) = E(\phi(n_1,T)) = 2 - \frac{K}{P^{n_2}} + \phi(n_1,T)(\frac{1}{P^{n_2}} - \frac{1}{P^{n_1}})$$

Le problème ainsi posé entraîne $\phi(n_1,T) \in [K - P^{n_2}, P^{n_1}]$: au maximum n_1 aura P^{n_1} salles, et au minimum ce que lui laisse n_2 quand elle a P^{n_2} salles. On remarque que $E(\phi)$ est linéaire, donc son minimum est atteint à une des bornes du domaine de ϕ.

Dès lors on cherche dans quelle condition $E(K - P^{n_2}) > E(P^{n_1})$. Le calcul montre que cette condition est satisfaite si $P^{n_2} > P^{n_1}$. On conclut, que l'équipe demandant le moins de salles est satisfaite au maximum. On généralise en considérant plus d'équipes. Les équipes sont d'autant mieux servies en salles qu'elles demandent moins de salle.

Le graphique suivant montre une répartition de salles pour cinq équipes qui ont demandé respéctivement 10, 10, 11, 12, et 13 salles, pour un campus comptant 24 salles. Les équipes 1 et 2 ont été servies complètement, l'équipe 3 a pris le reste, et les deux autres équipes n'ont rien obtenu. Les coefficients de préférence pour toutes les équipes sont tous à 1. Aucune

autre contrainte ne rentre en jeu.



(+) : équipe 1, (×) : équipe 2, (⊕) : équipe 3.

Cette répartition n'est pas satisfaisante. La fonction d'erreur 1 doit être revue.

C.1.2 Confrontation (nombre de salles désirées) - (Distance à bâtiment)

On se pose ici la question suivante : dans quelles conditions l'ajout d'une salle réelle à une équipe demandant à être proche d'un bâtiment provoque une baisse de l'erreur. En effet, si on ajoute une salle à une équipe l'erreur liée à E_1 baisse, mais si la salle allouée n'est pas dans le bâtiment visé alors l'erreur liée à E_2 augmente.

Considérons une équipe n demandant un nombre de salles P^n et la proximité à un bâtiment b. Pour simplifier on note $\phi = \phi(n, T)$, et $\sum d$ la somme des distances au bâtiment

b des salles de l'équipe n'appartenant pas à b. De plus d_{max} est la distance maximum d'une salle du campus à b. Partant d'un état T caractérisé par l'erreur $E(T)$, on attribue une salle à n telle que sa distance d' à b soit minimum parmi les salles vides restantes. Le nouvel état est T'. La différence d'erreur s'écrit :

$$E(T)-E(T') = E_1^n(T)-E_1^n(T')+E_2^n(T)-E_2^n(T') = \frac{P^n-\phi}{P^n}-\frac{P^n-\phi-1}{P^n}+\frac{\sum d}{\phi.d_{max}}-\frac{\sum d+d'}{(\phi+1).d_{max}}$$

L'erreur décroît si :

$$d' < \frac{\sum d}{\phi.d_{max}} + \frac{(\phi+1).d_{max}}{P^n}$$

Dans cet exemple, pour que l'échange entre une salle virtuelle et une salle réelle ait lieu, il faut que la distance d' séparant cette salle réelle du bâtiment vérifie cette inégalité. On peut illustrer cette propriété en prenant une instance à deux bâtiments trés éloignés l'un de l'autre.

Le comportement de la fonction d'erreur liée au nombre de salles désirées est à nouveau mis en défaut : il est plus intéressant d'avoir une salle éloignée que pas du tout.

C.1.3 Confrontation (distance à bâtiment) - (distance à bâtiment)

On étudie le cas où les deux équipes n_1 et n_2 demandent seulement à être dans le même bâtiment b. Le bâtiment b dispose de K^b salles. Les deux équipes veulent respectivement P^{n_1} et P^{n_2} salles. On envisage le cas où les besoins en nombre de salles sont satisfait pour les deux équipes. Dans ce cas seule l'erreur liée à la distance au bâtiment existe. Supposont que $P^{n_1} \leq K^b$, $P^{n_2} \leq K^b$, et $(P^{n_1}+P^{n_2}) > K^b$. Supposont également que l'équipe n_1 occupe intégralement b et que les salles allouées à l'équipe n_2 minimisent les distances à b. Alors si T est l'affectation, l'erreur globale s'écrit :

$$E(T) = E_2^2(T) = \frac{\sum_{i\in\Phi^b(n_2,T)} d_{ib}}{\phi(n_2,T)}$$

Calculons maintenant l'erreur si on échange une salle de n_1 avec une salle de n_2 dans le cas où cette salle n'est pas dans b. On suppose que cette dernière salle est distante du bâtiment b d'une distance d.

L'erreur globale s'écrit alors :

$$E'(T) = E_2^1(T) + E_2^2(T) = \frac{d}{\phi(n_1,T)} + \frac{-d+\sum_{i\in\Phi^b(n_2,T)} d_{ib}}{\phi(n_2,T)}$$

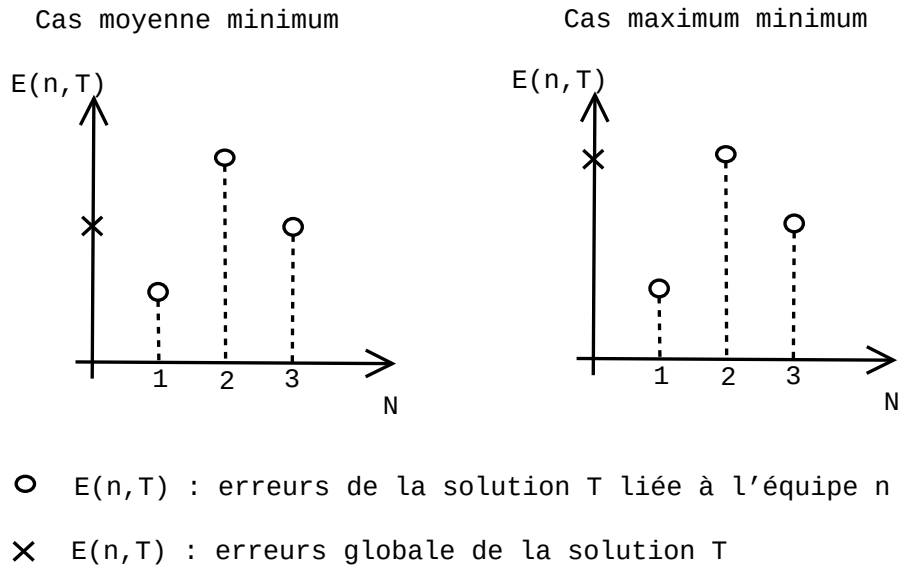
Par rapport à $E(T)$, l'erreur $E'(T)$ suite à l'échange sera :
- égale si les deux équipes ont le même nombre de salles attribuées,
- inféreure si $\phi(n_1,T) \geq \phi(n_2,T)$,
- supérieure sinon.

Donc dans notre cas, l'équipe qui demande le moins de salles est prioritaire pour occuper le bâtiment visé par les deux équipes. Donc, au pire, si $\phi(n_1, T) \leq \phi(n_2, T)$, et si $P^{n_1} = K^b$ alors le système affectera toutes les salles de b à n_1 et placera n_2 dans le bâtiment le plus proche de b. Plus généralement, c'est l'équipe qui a le moins de salles attribuées qui est prioritaire. Dans le cas où il y a égalité, il y a stabilité quelque soit la répartition des salles.

Cette répartition n'est pas satisfaisante car elle ne tient pas compte des préférences posées, supposées égales entre les équipes. Dans ce cas il serait plus convenable que le système fasse une répartition dans le bâtiment visé proportionnelle au nombre de salles demandées par les équipes. Dans le cas de préférences différentes, l'équipe posant le coefficient le plus élevé devrait être favorisé sans pénaliser tout à fait l'autre équipe.

C.2 Cas 'Maximum Minimum'

Il s'agit à nouveau d'étudier le comportement relatif des contraintes lorsqu'elles sont mises en concurence, mais avec une nouvelle fonction objectif qui cherche à minimiser la plus grande insatisfaction des équipes.



L'objectif est de chercher à insatisfaire le moins possible l'ensemble des équipes, alors que la méthode 'moyenne minimale' peut accepter de grandes disparités, pourvu que la moyenne soit basse.

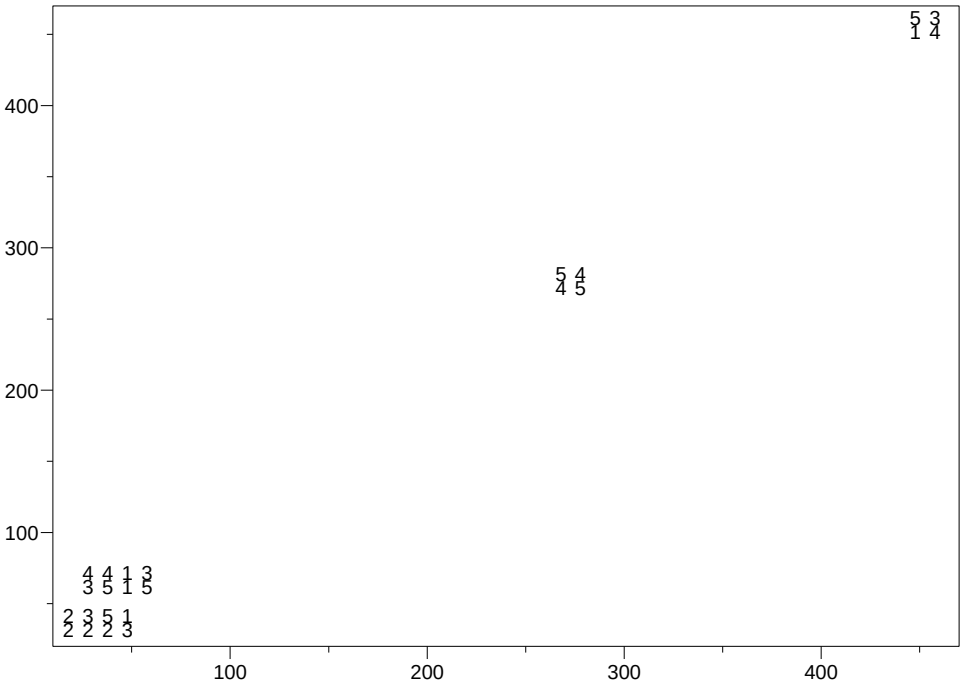
Nous allons voir que cette méthode garantit une meilleure répartition des ressources sans pour autant changer les fonctions d'erreur. L'algorithme utilisé précédemment pour la méthode 'moyenne minimale' a été revu pour garantir des erreurs minimales pour toutes les équipes. En effet, si on se contente de minimiser uniquement l'erreur de l'équipe la plus insatisfaite, on peut obtenir des solutions où il n'est plus possible de faire décroître l'erreur maximale, et où l'erreur minimale pour les autres équipes n'est pas atteinte. Pour cela l'algorithme utilise deux listes tabous simples, une sur les variables de décision liées à l'équipe traitée à un instant donné, et une autre liste sur les équipes afin de ne pas sélectionner une équipe pour laquelle toutes les variables sont tabous.

Les tests réalisés dans le cas 'moyenne minimale' sont repris pour montrer l'évolution par rapport à l'autre méthode.

C.2.1 Confrontation (nombre de salles désirées) - (nombre de salles désirées)

Le graphique suivant montre une répartition de salles pour cinq équipes qui ont demandé respéctivement 10, 10, 11, 12, et 13 salles, pour un campus comptant 24 salles.

Avec la méthode 'moyenne minimale', les équipes demandant le moins de salles étaient servies en totalité. Ici les salles sont réparties au prorata des demandes et de la place disponible dans le campus. Les équipes recoivent respectivement 4, 4, 5, 5, et 6 salles. Aucune autre contrainte ne rentre en jeu.



La bâtiment 1 est en bas à gauche. Le 4 est en haut à droite. Les autres sont dans l'ordre.

Cette fois la répartition est satisfaisante.

C.2.2 Confrontation (nombre de salles désirées) - (Distance à bâtiment)

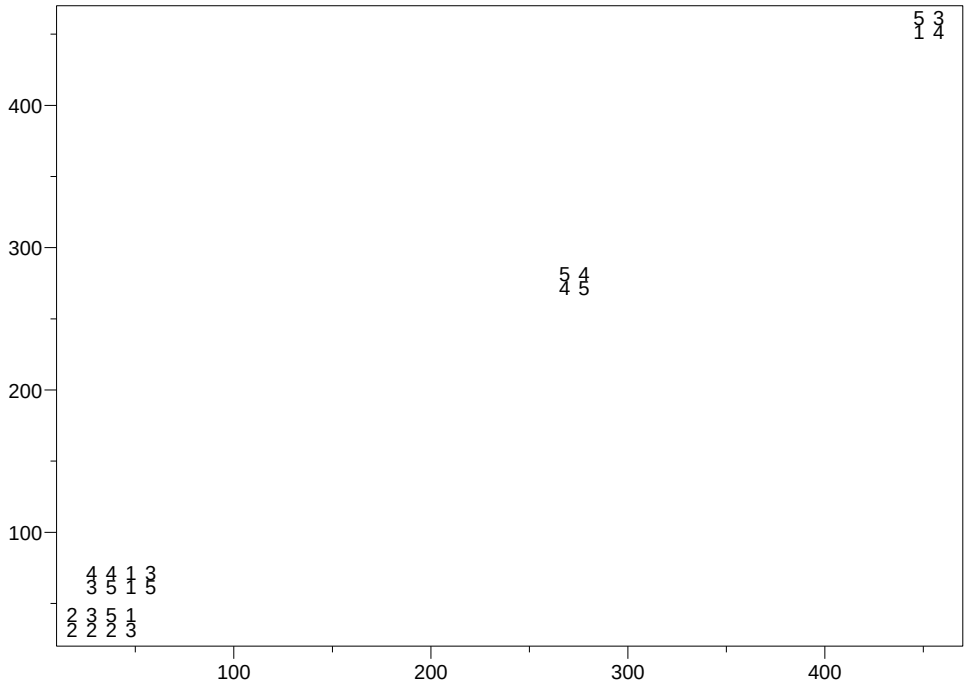
Ne faisant intervenir qu'une équipe, cette étude reste valable ici et le cas peut se présenter où une équipe ne sera pas complètement servie en salle alors qu'il reste de la place. Cependant ce cas particulier peut être traité par le double mécanisme de restart, et d'affectation initiale aléatoire.

C.2.3 Confrontation (distance à bâtiment) - (distance à bâtiment)

C'est l'équipe qui demande le plus de salles qui est le mieux servie. Plus une équipe demande de salles, plus elle doit etre logée dans un bâtiment éloigné de celui demandé, ce

qui engendre un cout élevé sur les salles éloignées, et qui va faciliter l'affectation des autres salles de l'équipe dans le batiment visé. L'autre équipe occupe alors le bâtiment le plus proche de celui visé.

Le comportement de cette fonction d'erreur est correct et on peut considérer les deux équipes également satisfaites.



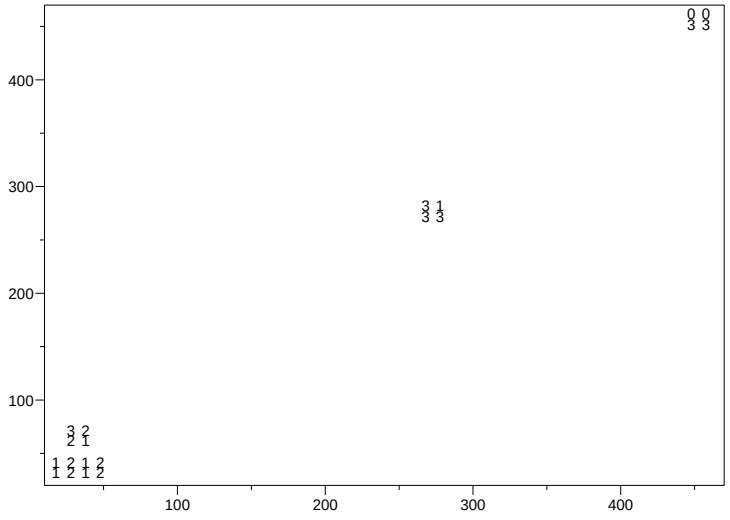
L'influence du coefficient de préférence d'une équipe sur la contrainte est vérifiée. Une equipe posant un coefficient plus fort que les autres pour cette contrainte sera plus satisfaite.

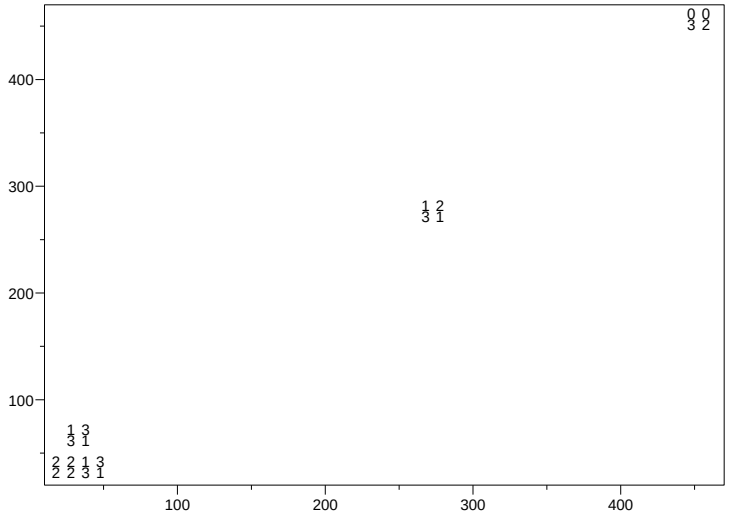
C.2.4 Confrontation (distance à bâtiment) - (distance à équipe)

Cette confrontation n'a pas été étudiée finement avec $v4$. Cependant les tests montrent que l'équipe $n1$ qui demande à être proche de l'équipe $n2$ est satisfaite dés lors qu'une seule salle de $n2$ est dans le même bâtiment que $n1$. De là on peut avoir des situations où $n2$ satisfait toutes ses contraintes sauf pour une salle, intégrée dans le même bâtiment que $n1$.

Avec la version 'maximum minimal', le comportement est le même qu'avec la version 'moyenne minimale'. Il est possible de corriger un peu cette contrainte en ne considérant plus les distances minimum avec les autres salles de l'équipe $n2$ mais la moyenne des moyennes des distances des salles de $n1$ à celles de $n2$. Alors, $n1$ sera mieux imbriquée avec $n2$, mais $n2$ sera plus pénalisée quant à sa distance à bâtiment.

Illustration : 3 équipes, $n1$ et $n2$ veulent $b1$ (en bas à gauche), $n3$ veut être proche de $n1$.


400
300
200
100
100
200
300
400


400
300
200
100
100
200
300
400

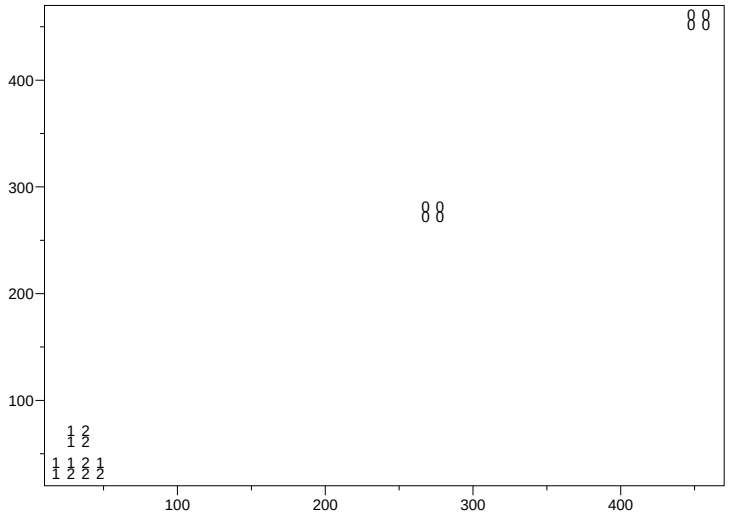
Comportement des coefficients : si $n2$ augmente son coefficient de préférence sur la contrainte distance à bâtiment elle gagne 1 salle dans b_1 pour $\alpha_1 = .2$ et $\alpha_2 = .8$.

L'équipe qui pose la contrainte va 'tirer' l'autre équipe à elle.

C.2.5 Confrontation (distance à bâtiment) - (dispersion)

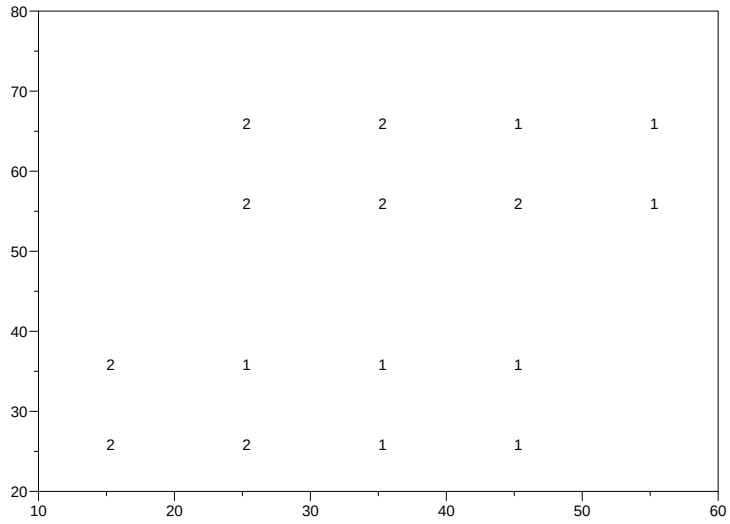
On considère le cas où deux équipes veulent de façon identique être dans le même bâtiment, et veulent être regroupées.

Les résultats obtenus montrent un bon comportement de cette contrainte, i.e. les salles se regroupent. Cependant la géométrie particulière du campus simple utilisé ne permet pas toujours d'obtenir des résultats cohérents, du fait de sa symétrie. On peut arriver à des situations d'équilibre non satisfaisantes (salles isolées) de même erreur sur les équipes qu'une solution plus convenable.



Afin d'essayer de briser la symétrie, on peut ajouter 4 salles à b_2.

Exmple suivant : mêmes exigences, mêmes coefficients. L'erreur minimale est atteinte. L'insatisfaction de n_2 liée à E_2 est compensée par le regroupement. Les équipes sont séparées en deux parties égales du fait de la distance à bâtiment : plus grande de la droite de b_2 à b_1 que de la gauche de b_2 à b_1.



Influence des coefficients : mauvaise car plus n_1 demande à être regroupée, plus il a de salles dans le bâtiment visé au détriment de n_2. Cas $\alpha_2^{n_1} = .1$, $\alpha_4^{n_1} = .9$, $\alpha_2^{n_2} = .9$, $\alpha_4^{n_2} = .1$:

La distance utilisée par E_4 est une distnace maximale entre deux salles alors que pour E_2 c'est une distance moyenne. De fait ça marche mieux en prenant pour E_4 une distance moyenne à minimiser comme critere de dispersion. Pour une contrainte la valeur moyenne est mieux qu'une valeur maximale car dans la cas où deux variables renvoient des valeurs maximales identiques alors l'algorithme stagne. Pour E_2 il faut conserver des projections nulles pour les salles du bâtiment b attribuées à une équipe qui demande ce bâtiment, car sinon l'algorithme affecte plus facilement des salles dans les bâtiments proches de celui visé, alors qu'il reste de la place dans ce dernier. Ceci est dû au fait qu'on prend comme distance à un bâtiment la distance de Manhatan entre la salle et le centre du batiment (moyenne des coordonnées Consensus). Il serait plus judicieux de prendre en compte les accès aux bâtiments pour le calcul des distances.

C.2.6 Confrontation (distance à bâtiment) - (dispersion)

Contraintes non concurentes. Comportements conformes.

D Trace de l'exécution étudiée au chapitre 6 : évaluations

```
calcul numero 1, debut :
Tue Jun 21 10:24:41 CEST 2005
WARNING:En calcul ...
initialisation glouton
Restart, iteration : 557
Restart, iteration : 1043
Restart, iteration : 1529
Dernière iteration : 1971
WARNING:Notation par client
Ajout de la solution : 20051721227001
Ajout de la solution : 20051721227002
Ajout de la solution : 20051721227003
Ajout de la solution : 20051721227004

calcul numero 2, debut :
Tue Jun 21 12:27:57 CEST 2005
WARNING:En calcul ...
initialisation glouton
Restart, iteration : 496
Restart, iteration : 1205
Restart, iteration : 1664
Dernière iteration : 2255
WARNING:Notation par client
Ajout de la solution : 20051721522001
Ajout de la solution : 20051721522002
Ajout de la solution : 20051721522003
Ajout de la solution : 20051721522004

calcul numero 3, debut :
Tue Jun 21 15:22:39 CEST 2005
WARNING:En calcul ...
initialisation glouton
Restart, iteration : 520
Restart, iteration : 999
Restart, iteration : 1440
Dernière iteration : 1955
WARNING:Notation par client
Sorry, TK has not been enabled this the session.
Ajout de la solution : 20051721735001
Ajout de la solution : 20051721735002
Ajout de la solution : 20051721735003
Ajout de la solution : 20051721735004

calcul numero 4, debut :
Tue Jun 21 17:35:53 CEST 2005
WARNING:En calcul ...
initialisation glouton
Restart, iteration : 521
Restart, iteration : 997
Restart, iteration : 1491
Dernière iteration : 2034
WARNING:Notation par client
Ajout de la solution : 20051721941001
Ajout de la solution : 20051721941002
Ajout de la solution : 20051721941003
Ajout de la solution : 20051721941004
```

```
secherre@rochegude:~/public_html/AS$ ./gogo.sh
calcul numero 1, debut :
Wed Jun 22 07:49:25 CEST 2005
WARNING:En calcul ...
initialisation glouton
Restart, iteration : 611
Restart, iteration : 1184
Restart, iteration : 1750
Dernière iteration : 2233
WARNING:Notation par client
Ajout de la solution : 20051731025001
Ajout de la solution : 20051731025002
Ajout de la solution : 20051731025003
Ajout de la solution : 20051731025004

calcul numero 2, debut :
Wed Jun 22 10:26:07 CEST 2005
WARNING:En calcul ...
initi{scriptsize}alisation glouton
Restart, iteration : 502
Restart, iteration : 996
Restart, iteration : 1437
Dernière iteration : 1921
WARNING:Notation par client
Ajout de la solution : 20051731253001
Ajout de la solution : 20051731253002
Ajout de la solution : 20051731253003
Ajout de la solution : 20051731253004

calcul numero 3, debut :
Wed Jun 22 12:54:08 CEST 2005
WARNING:En calcul ...
initialisation glouton
Restart, iteration : 486
Restart, iteration : 980
Restart, iteration : 1421
Dernière iteration : 1904
WARNING:Notation par client
Ajout de la solution : 20051731500001
Ajout de la solution : 20051731500002
Ajout de la solution : 20051731500003
Ajout de la solution : 20051731500004

calcul numero 4, debut :
Wed Jun 22 15:01:15 CEST 2005
WARNING:En calcul ...
initialisation glouton
Restart, iteration : 486
Restart, iteration : 993
Restart, iteration : 1434
Restart, iteration : 2005
WARNING:Notation par client
Ajout de la solution : 20051731722001
Ajout de la solution : 20051731722002
Ajout de la solution : 20051731722003
Ajout de la solution : 20051731722004
```

E L'apprentissage des préférences

Dans l'exemple qui suit, une équipe demande des salles de types différents (les quatre premières colonnes : type 1 = 0, type 2 = 5, type 3 = 10, type 4 = 0). Elle peut demander une distance par rapport à deux lieux (ici proche du bâtiment 25), et à deux équipes (aucune), enfin l'équipe demande à être regroupée.

Le vecteur des paramètres s'écrit alors :

$$PP = [0, 5, 10, 0, 25, 0, 0, 0, 1]$$

Le vecteur des coefficients associés est :

$$\alpha = [0, 0.2, 0.2, 0, 0.2, 0, 0, 0, 0.4]$$

Les préférences réelles de l'équipe sont connues du logiciel afin de pouvoir tester l'algorithme d'apprentissage, mais elles ne servent qu'au moment de calculer les notes des solutions. (fichier `ParametresLoop.sce`) Ces préférences sont identiques à celles initialement posées, sauf que l'équipe ne désire pas être proche du bâtiment 25, mais loin de l'équipe 1, avec le même coefficient.

Aprés calcul et épuration, on obtient :

$$PP = [0, 0, 0, 0, 0, 0, 10, -3, 0]$$

$$\hat{\alpha} = [0, 0, 0, 0, 0, 0, 0.176343, 0.176343, 0]$$

Enfin, aprés agrégation, et normalisation :

$$PP = [0, 5, 10, 0, 25, 0, 10, -3, 1]$$

$$\hat{\alpha} = [0, 0.148072, 0.148072, 0, 0.148072, 0, 0.130557, 0.129083, 0.296144]$$

Le calcul de cette première évaluation porte sur quatre notations.

L'analyse des préférences des autres utilisateurs montre que l'équipe 3 demande à être proche de l'équipe 1, dont l'éloignement est secrètement désiré par l'équipe joueuse. S'éloigner de l'équipe 3 revient donc à s'éloigner de l'équipe 1. En revanche la proximité souhaitée de l'équipe 10 est plus douteuse car de par ses souhaits, l'équipe 10 sera proche de la région où l'équipe 1 a demandé à être. Cependant, et malgré cette incohérence, les solutions proposées sont susceptibles de satisfaire l'équipe joueuse.

A noter qu'une solution fait cohabiter cette équipe avec celle dont il veut secrètement être éloignée. Cette solution, qui est un minimum local, est utile à la résolution des prochains calculs. La mauvaise note qui en découle permet de donner des informations à la programmation linéaire.

L'itération suivante donne les vecteurs suivants, après épuration :

$$PP = [0, 5, 10, 0, 26, 27, -1, -3, 0]$$

$$\hat{\alpha} = [0, 3.815796, 0, 0, 0.262420, 0.135100, 0.172843, 0.080546, 0]$$

Aprés agrégation, et normalisation :

$$PP = [0, 5, 10, 0, 25, 26, -1, -3, 1]$$

$$\hat{\alpha} = [0, 0.131943, 0.131943, 0, 0.131943, 0.262420, 0.114027, 0.053137, 0.2638850]$$

On note la disparition de la proximité à l'équipe 10, et l'apparition de l'éloignement à l'équipe 1, comme désiré. On note sans l'expliquer l'apparition d'une contrainte de proximité au bâtiment 26 avec un coefficient relatif fort. Cette contrainte contribue à l'éloignement recherché.

Références

[1] www.inria.fr, 2005.

[2] Phillipe CODOGNET and Daniel DIAZ. Yet another local search method for constraint solving. *SAGA01*, 2001.

[3] François FAGES, Evelyne LUTTON, Martin PERNOLLET, and Matthieu PIERRES. Consensus : a hybrid system for multi-user decision-aid in space allocation problems. May 7,2004.

[4] Claude GOMEZ. *Engineering and Scientific Computing with Scilab.* Birkhäuser, 1999.

[5] SAMARIN GOOSSENS, MITTELBACH. *The Latex companion.* Addison-Wesley, 1994.

[6] Martin PERNOLLET. Rapport de fin de stage. May 7,2004.

[7] Larry WALL, Tom CHRISTIANSEN, and Randal L. SCHWARTZ. *Programming Perl.* O'Reilly, 1996.

Table des matières

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803