École Internationale des Sciences du Traitement de l'Information

Département Informatique

Projet GSI

Shable - Gestionnaire de placement en examen

Auteur :
Marc BRAMAUD DU BOUCHERON
Christophe GUILLONNET

Encadrant :
Bernard Glonneau

2014

Table des matières

Introduction

Dans le cadre du semestre 4 (ING2) de nos études à l'EISTI, nous avions pour projet de réaliser une application permettant de placer des élèves de façon optimale dans une salle d'examen.

L'énoncé du devoir expose notamment les contraintes suivantes :

— pouvoir placer `n` élèves dans une salle contenant `p` tables indépendamment de `n` et `p` ;
— être capable de *quantifier le risque de triche* entre élèves ;
— mettre à profit les compétences acquises dans le module de programmation et d'algorithmique parallèle.

Afin de répondre à ce problème, il nous a donc été nécessaire de réunir nos connaissances en développement, mais aussi en algorithmique. En effet, si ce problème peut sembler simple au premier abord, il est en réalité assez complexe.

Complexité du problème

Dans cette partie introductive, nous retranscrirons notre analyse initiale du problème avant d'analyser les conséquences algorithmiques qui ont découlé.

Considérons le cas trivial suivant : comment placer 4 élèves dans une salle rectangulaire avec `p` tables réparties equitablement ?

Cette question est totalement triviale. de même, on peut aisément augmenter le nombre d'élèves légèrement et arriver à des solutions élégantes.

Cependant, le problème devient rapidement beaucoup plus complexe lorsque l'on change la géométrie de la salle ou que l'on augmente grandement le nombre d'élèves (essayer de placer 20 élèves dans une salle polygonale avec des tables réparties aléatoirement est loin d'être trivial !)

Recherche d'un algorithme

Avant de chercher un algorithme, il est nécessaire de modéliser le problème. Voici nos hypothèses :

— quantification de la triche entre élèves : pour quantifier le risque de triche entre élèves, nous nous sommes basés sur la proximité entre élèves. Un groupe d'élèves rapprochés dans une salle aura plus de facilité à tricher que des élèves épars. Notre critère est donc le suivant : pour un élève donnée, son risque de triche est directement corrélé à la moyenne de ses distances aux autres élèves. Pour simplifier nos propos dans cette partie, nous appellerons cette moyenne M.
— objectif à atteindre : le but est donc ici de maximiser pour chaque élèves sa valeur de `M`.
— Globalement, nous voulons maximiser la valeur de `M` pour toute la classe.

Nous avons fait quelques recherches afin trouver un problème connu proche du notre. Toutefois, ce problème n'est pas un problème très connu. Des formes particulières du problème du voyageur de commerce se rapprochent de ce problème, mais nous n'avons pas réussi à exploiter celles-ci ; les connaissances mathématiques associées étaient hors de notre portée, et le développement de ces solutions aurait été très difficile.

Par ailleurs, ce problème est un problème `NP`-complet, sa solution ne peut donc pas être trouvé en un temps raisonnable.

Nous nous sommes donc tournés sur un algorithme que nous connaissons mieux, et qui donne des résultats honnêtes : l'algorithme du Recuit Simulé.

Chapitre 1

Algorithme

Notre algorithme est un recuit simulé dont l'énergie est la somme des moyennes des distances entre les tables occupées.

Le recuit simulé nécessitant une phase d'initialisation, nous avons commencé par obtenir une solution initiale la moins mauvaise possible.

Ce chapitre détaille le déroulement de l'algorithme utilisé en pseudo code.

1.1 Initialisation

En partant d'une salle et d'une classe, on place les élèves un par un le plus loin (en moyenne) de ceux placés auparavant.

Exemple : dans le cadre d'une salle carré, et en placant notre premier élève dans le coin supérieur gauche, notre algorithme d'initialisation donnera la table du coin inférieur droit comme prochain emplacement. Le suivant sera soit le coin supérieur droit, soit le coin supérieur gauche. (Nous déterminons en fait l'emplacement qui maximise `M`)

1.2 Choix de conception

Comme nous l'avons vu précedement, nous nous intéressons plus à l'emplacement des élèves qu'aux élèves eux même. Notre modèle n'est donc pas *très* proche de la réalité.

En résumé, voici la composition de notre programme :

— une classe `Master`, qui applique notre algorithme de Recuit Simulé ;
— une classe `Eleve`, qui symbolise simplement un élève ;
— une classe `Classe`, qui contient simplement une collection d'objets `Eleve` ;
— une classe `Table`, second coeur de notre programme ;
— enfin, une classe `Salle`, qui a principalement pour d'effectuer des actions sur sa propre collection de table ;
— enfin, une classe `Createur`, qui délocalise la création de notre classe et et de notre `Salle`.

1.2.1 Classe `Master`

La classe `Master` est simplement notre `main()`. C'est elle qui instancie nos `Eleve` et forme ensuite une `Classe`, puis instancie nos `Table` et forme ensuite une `Salle`. Cela effectué, c'est la classe `Master` qui via la classe `Classe`, implémente l'algorithme de Recuit Simulé.

1.2.2 Classe `Table`

C'est ici que réside les difficultés que nous avons rencontrées. Etant donné que nous nous intéressons aux `emplacements` de nos élèves, c'est donc la classe `Table` que nous avons le plus développé.

Nous précedement établit que nous choisisons un futur emplacement pour un élève en fonction de sa distance aux autres élèves. Cela se traduit par : "nous choisissons parmis les tables libres celle dont

les distances aux tables occupées sont les plus grandes". En utilisant notre notation `M`, cela signifie que nous choisissons parmis les tables libres celles qui présentent un `M` le plus élevé.

Pour calculer ce `M` pour chaque table de nombreuses reprises dans notre algorithme, nous avons décidé de mettre en pratique le multi-threading. Notre classe `Table` implémente donc l'interface `Runnable`. Chaque table est donc sensée calculer elle-même sa valeur de `M`, parmis d'autres opérations. Pour cela, il est nécessaire que l'execution de ces méthodes provienne de la méthode `run()` de la table.

Il est illégal de lancer plusieurs fois la méthode `start()` sur un Thread, il est donc nécessaire de trouver un moyen de conserver la méthode `run()` vivante, et que celle-ci puisse effectuer des actions différentes. Voici nos solutions :

- utilisation d'une énumération `Ordre` : il s'agit de la liste d'ordre que peut executer une table ;
- utilisation des méthodes `wait()` et `notify()` : l'appel de la méthode `wait()` sur un objet "Declencheur" (attribut commun à toutes nos tables), permet d'endormir nos `Tables`(Threads) à la fin de l'execution d'un ordre. C'est notre solution de mise en veille de nos threads. Lorsque `Master` (via Salle) appelle la méthode `notifyAll()` sur ce même objet, nos `Tables` se réveillent et executent l'ordre présent dans leur énumération.
- utilisation d'une sémaphore : dans notre classe `Master`, le lancement d'opérations dans nos threads nous rends la main immédiatement. `Master` execute donc des instructions dépendantes de résultats que nos tables n'ont pas fini de calculer. Pour pallier à ce problème, Master essaye d'acquérir `n` jetons sur un objet `Sémaphore`, jetons qui sont relachés par nos `n` tables quand elles ont finis leurs calculs uniquement. Cette sémaphore agit donc comme un élément bloquant l'execution du thread principal `Master` tant que nos calculs répartis dans nos tables ne sont pas terminées.
- un objet déclencheur, commun à toutes nos

nous avons premièrement pensé inclure un attribut `Classe` dans nos `Tables`, permettant ainsi à toutes tables de pouvoir acceder aux informations des autres tables (la table voisine est-elle occupée ou non ? Quelles sont ses coordonées ?). Toutefois cette solution présentait le problème d'un accès concurrent à notre instance de `Classe` et ses attributs (des tables qui se posent des questions).

```
PROCEDURE initialiser(Classe classe, Salle salle)
    Eleve e1 = classe(1)
    Table t1 = salle(1)

    // Mettre le premier eleve sur la premiere table
    t1.ajouterEleve(e1)

    POUR eleve DANS classe FAIRE
     // trouver la table libre la plus lointaine
     // des tables occupees (en moyenne).
     Table t = salle.tablesLibres.calculerTableMin

     // ajouter l'eleve courant à la table trouvee
     t.ajouterEleve(eleve)
    FIN POUR
    Energie e0 = salle.calculerEnergie

    RETOURNER e0
FIN PROCEDURE
```

1.3 Recuit Simulé

Une fois la solution initiale obtenue, on lui applique un algorithme de recuit simulé.

```
PROCEDURE optimiser(Salle salle, Classe classe)
// declaration des variables
```

```
Entier t = 100
Entier t_min = 75 // minimum a ajuster empiriquement
Entier compteur= 1
Float rand_rs = RANDOM%1
Entier rand_libre = (Entier) RANDOM
Table t = salle.tables_occupees(0)
salle s_temp = salle
Energie e0 = initialiser(salle, classe)
Energie e_temp = e0
Energie e_temp_1 = e0

// contient les probabilites de choisir une table parmi
// les 20% d'energie individuelle max
Tableau de Entier tab_proba =
         Tableau de Entier[0.2*salle.tables_occupees.taille]

tab_proba[0] = 0.5 // probabilité de 0.5 de bouger la table
                   // dont l'energie est maximale


TANT QUE compteur<tab_proba.taille FAIRE
    tab_proba[compteur] = tab_proba[compteur-1]/2
    compteur += 1
FIN TANT QUE

TANT QUE t>t_min FAIRE
    rand = RANDOM%1
    salle.tables_occupees.ordonnerParEnergie
    compteur = 0

    // selectionner la table correspondant la proba
    // calculee
    TANT QUE tab_proba[compteur] < rand FAIRE
        compteur += 1
    FIN TANT QUE

    // changer l'eleve selectionne de place
    rand_libre = RANDOM%salle.tables_libres.taille

    s_temp.tables_libres(rand_libre).mettreEleve(
                    salle.tables_occupees(compteur))

    s_temp.tables_occupees.enleverEleve(
                    salle.tables_occupees(compteur))

    e_temp_1 = s_temp.calculerEnergie

    SI e_temp_1 < e_temp FAIRE
        salle = s_temp
    SINON FAIRE
        rand = RANDOM%1
        SI rand > expt( - (e_temp_1 - e_temp)/ t ) ALORS
            e_temp = e_temp_1
```

```
                salle = s_temp
                t = 0.95*t
            FIN SI
        FIN SI
    FIN TANT QUE

    RETOURNER salle

FIN PROCEDURE
```

Chapitre 2

Code

2.1 Technologies utilisées

Nous avons utilisé pour la mise en pratique de l'algorithme `Java 1.7`, notamment les méthodes de programmation parallèle telles que l'interface `Runnable`.

Ce code est integré dans un projet `Java EE` tournant sur `tomcat` et utilisant le framework front-end `Foundation 5`. L'affichage de nos tables n'est pas graphique, mais dans un tableau HTML. Concernant la base de données d'élèves et de salle, celle-ci est inscrite en dure dans notre programme, par manque de temps. Nous avions envisagés l'utilisation de SQLLite, qui n'est pas très difficile d'accès, mais là aussi n'avons pas eu le temps de l'implémenter, due à une mauvaise gestion de notre temps.

2.2 Exemples de difficultés rencontrées

2.2.1 Réveil des threads `Table`

Lorsque les objets `Table` (qui implémentent `Runnable`) sont réveillés afin d'obtenir la solution initiale, il intervient une *race condition* entre le thread principal et ses fils.

En effet, chaque thread `Table` effectue des calculs .

Dans la première version de *Shable*, nous n'avions pas géré ce problème et nous nous sommes retrouvés dans la situation suivante : lors de la phase d'initialisation (*cf.* 1.1), l'ensemble des threads fils obtenait lors de son réveil l'état de la salle et commençait ses calculs, mais le thread père continuait sa route et lançait l'itération suivante sans attendre les résultats. Par conséquent nous nous retrouvions avec une solution de départ (en fonction du temps d'éxecution de chaque thread) qui mettait 50% des élèves dans un coin de la salle et le reste dans l'autre.

Nous avons résolu ce problème en ajoutant à la classe `Master` un attribut de type `Semaphore` qui a comme nombre de *token* exactement le nombre de `Table` dans la salle. Avant de réveiller les threads, la classe `Master` réserve l'ensemble de ces *token* qui sont ensuite libérés un par un par ses fils. `Master` doit alors attendre que l'ensemble des *token* soit libre (via une nouvelle réservation) avant de continuer.

Ainsi, la classe `Master` attend effectivement la fin de l'ensemble des calculs des différentes tables avant de sélectionner la suivante lors de l'initialisation.

2.2.2 L'étrange bug du double `notifyAll`

Lors de l'appel de la fonction `declencher` de la classe `Table`, les threads correspondants sont réveillés par un appel de la fonction `notifyAll`.

Au moment du développement, nous avons passé *beaucoup* de temps [1] à essayer de résoudre un bug étrange : l'appel de cette fonction pourtant simple réveillait les threads concernés, sauf le tout premier.

A ce jour, nous ne connaissons toujours pas l'origine de ce comportement. Cependant nous avons fini par faire fonctionner la fonction en appelant deux fois de suite la fonction `notifyAll`.

1. 6h, entre 22h et 4h un jour de semaine

Chapitre 3

Améliorations potentielles

3.1 Algorithmiques

Nous pensons qu'il est possible d'améliorer notre algorithme en introduisant notamment les notions suivantes :

- proposer différentes méthodes de calcul de la solution initiale afin d'améliorer les performances du recuit. Ceci est notamment possible en différenciant les cas en fonction de différents critères (liste non exhaustive) :
  - taille de la classe ;
  - taille de la salle ;
  - symmétrie de la salle ;
  - rapport d'ordre de grandeur entre la taille de la salle et celle de la classe.
- améliorer l'efficacité de la seconde partie de l'algorithme par l'utilisation de grandes quantités de résultats empiriques (la température initiale ainsi que sa fonction de décroissance sont des choix arbitraires qui peuvent influencer la qualité des résultats et le temps d'éxecution) ;
- considérer plus de tables occupées potentiellement modifiables. Dans la version actuelle, seules les 20% ayant les moyennes de distance les plus hautes sont candidates au mouvement, les autres étant fixes. De plus celles-ci sont sélectionnées avec une probabilité décroissante (50%, 25%, ...) ;
- changer la méthode de sélection de la table vide sur laquelle l'élève sélectionné est placé. Cela permettrait notamment de considérer des voisins qui, si ils sont *a priori* plus mauvais peuvent s'avérer judicieux plusieurs itérations après ;
- déplacer plusieurs élèves à chaque itération. Cela permettrait (de même que la proposition précédente) de considérer un espace de solution plus important par conséquent de s'approcher de puits d'énergie qui peuvent être compliqués à atteindre avec l'algorithme actuel ;
- ...

3.2 Technologiques

Dans le code, plusieurs améliorations de performance ou de modélisation nous semblent possible :

- optimiser l'utilisation de la programmation parallèle en incluant des calculs d'énergie décentralisés et/ou un accès en mémoire partagée lors du calcul de recuis (un thread représentant un voisin par exemple) ;
- conserver un historique des calculs (en incluant une base de données de résultats par exemple) et implémenter des algorithmes de *machine learning* afin de prévoir à l'avance quelle variation de l'algorithme utiliser ;
- ...

3.3 D'interface

Nous pensons qu'il est crucial d'avoir une interface optimisée pour permettre à un utilisateur de comprendre les possibilités de l'application ainsi que les résultats des calculs effectués.

Pour cela, plusieurs solutions sont envisageables :

— afficher une représentation graphique de la solution. Nous avions consideré plusieurs options lors du développement de Shable, dont on peut d'ailleurs voir les restes dans les source (la bibliothèque `d3` dans `/WEB-INF/js`, le code `svg` commenté dans `/WEB-INF/private/dashboard.jsp`, ...) ;

— mettre à jour en direct sur l'interface web la progression du calcul et l'amélioration du potentiel de triche ;

— ...

Conclusion

Ce projet nous a donnée beaucoup de fil à retordre, essentiellement du fait de notre manque d'organisation temporelle et de notre difficulté à définir clairement les spécifications fonctionnelles et techniques. Nous avons pris énormément de retard et avons ensuite été embarqués dans d'autres occupations (nos stages notamment) qui ne nous ont pas facilité la tâche. Sur le plan de la gestion de projet et du respect des délais il s'agit donc d'un echec.

Nous retenons cependant certains points positifs sur différents plans.

Tout d'abord, nous réalisons à quel point il est impossible de se lancer tête baissée dans un projet techniquement compliqué et volontairement libre sur les technologies et algorithmes à utiliser. En tant que développeurs, nous avons ainsi beaucoup appris sur l'importance d'un cahier des charges bien défini, chose que nous retrouvons aujourd'hui dans nos différents stages.

De plus, nous avons eu grâce à Shable l'occasion de mettre en pratique les différents cours vu en 2^{me} année du cycle ingénieur : algorithmique et programmation parallèle, développement `Java EE`, Design Pattern, ...

Enfin, d'un point de vue personnel, nous avons découvert les difficultés nous poussent à vouloir faire les choses bien (nous sommes repartis de zéro à divers états d'avancement plus de cinq fois et utilisé trois langages différents avant de nous fixer sur `Java`) et à repenser en permanence l'architecture de notre projet. Bien qu'il s'agissent d'une sorte de variation sur le développement agile vu au semestre 3, nous avons compris que l'on revient presque naturellement à ces méthodes.

LaTeX 2_ε