Génie Informatique
Rapport de Travaux Pratiques

INF4705 : Lab 3

Résolution de problème combinatoire

Auteur
Eric Ah-Tiane
1760376
Gwenegan Hudin
1756642

Rendu
5 Décembre 2014
À Polytechnique Montréal

Table des Matières

I. Introduction

Le présent rapport documente les procédures utilisées lors du troisième laboratoire du cours INF4705 Analyse et Conception d'algorithmes.

Le laboratoire consiste à implémenter un algorithme d'optimisation afin de résoudre un problème combinatoire. Il faut minimiser les pertes liées à l'affectation des coupes dans les blocs de marbre. Nous avons donc un immense espace de solutions potentielles à explorer, parmi lesquelles de solutions optimales locales et une solution optimale globale.

Le choix de l'algorithme était laissé au choix mais il a été décidé d'utiliser un algorithme vorace randomisé avec amélioration locale. Cette méthode permet de déterminer rapidement plusieurs solutions optimales locales. Parmi celles-ci, il est possible, au bout d'un nombre d'essais assez grand, de trouver l'optimum global de l'espace de solutions.

Les sections suivantes présentes l'algorithme implanté et son analyse ainsi que l'originalité de celui-ci. Cet algorithme sera en compétition avec ceux des autres groupes du cours, et se verra attribuer seulement 3 minutes pour trouver la meilleure solution possible, après quoi son fonctionnement sera interrompu. Le temps d'exécution est donc important.

Des ensembles de données nous sont fournies avec 2, 3, 4, 5 ou 9 catégories de blocs.

II. Présentation de l'algorithme

1. Fonctionnement et pseudo-code

Lors de notre remue-méninges initial, nous avons considéré l'utilisation de la programmation par contraintes, particulièrement adaptée pour résoudre les problèmes combinatoires. Nous voulions utiliser le cadriciel Choco avec Java. Cependant, la programmation par contraintes sort de l'objectif de ce cours, et nous nous sommes donc rabattus sur la solution préconisée avec les outils à notre disposition. Nous avons implémenté en Java un algorithme vorace randomisé, avec une heuristique d'amélioration locale, et répétons ces opérations pour trouver différents optimums locaux. De ces optimums, nous gardons le meilleur, qui peut être l'optimum global (mais ce n'est pas garanti). Voir Figure 2.

```
function ResoudreProbleme(P)
    s_opt ← {}
    p ← ∞
    while true do
        s ← voraceRandomise(P, α)
        s ← ameliorationLocale(s)
        if Perte(s_opt) < Perte(s) then
            s_opt ← s
            afficher(s)
        end if
    end while
end function
```

Figure 2 – Pseudo-code de l'algorithme global

Un vorace non randomisé obtiendrait toujours la même solution, et amènerait donc toujours au même optimum local, qui n'est probablement pas l'optimum global du premier coup. On introduit donc un certain degré d'aléatoire en ne prenant non pas toujours la meilleure solution disponible, mais une des meilleures solutions disponibles, avec un intervalle généré par α. On obtient ainsi une liste réduite de candidats parmi lesquels on choisit aléatoirement. Voir Figure 3.

```
function voraceRandomise(P, α)
    s ← {}
    couts ← {}
    coupes ← initialiserCandidats(P)
    for all c ∈ coupes do
        couts[c] ← evaluer(c)
    end for
    for all c ∈ coupes do
        min ← minimum(couts)
        max ← maximum(couts)
        RCL ← {}
        borne ← max − α × (max − min)
        RCL ← {c ∈ coupes | cout[coupe] ≤ borne}
        e* ← random(RCL)
        ajouter(e*, s)
        miseAJour(coupes)
        miseAJour(couts)
    end for
    return s
end function
```

FIGURE 3 – Pseudo-code de la méthode vorace randomisée

Une fois que l'on a une solution vorace, on l'améliore localement par la méthode de descente de pente 1-opt. Cet algorithme intervertit une assignation de coupe aléatoirement au sein de la solution, et si elle est meilleure que la solution originale, alors il la garde et essaie à nouveau. Cet algorithme s'arrête lorsqu'il échoue un certain nombre de fois à la suite, ce qui indique qu'il n'a probablement plus aucun échange intéressant à réaliser, et donc qu'il a atteint l'optimum local. Ce nombre a été fixé à 30 empiriquement dans notre programme. Voir Figure 4.

```
function AMELIORATIONLOCALE(s_0)
    s_opt ← {}
    while essais < essais_max do
        bloc1 ← RANDOM(s_0)
        coupe ← RANDOM(bloc1[coupes])
        recepteurs ← {bloc ∈ s_0 | perte[bloc] ≥ coupe}
        bloc2 ← RANDOM(recepteurs)                          ▷ bloc2 ≠ bloc1
        TRANSFERERCOUPE(bloc1, bloc2)
        s_opt ← REMPLACER(s_0, bloc1, bloc2)
        REDUIRE(s_opt)
        if erreur ou VIDE(recepteurs) ou PERTE(s_opt) > PERTE(s_0) then
            essais++
        else
            essais ← 0
            s_0 ← s_opt
        end if
    end while
    return s_opt
end function
```

FIGURE 4 – Pseudo-code de la méthode d'amélioration locale 1-opt

2. Analyse de complexité

Soient n le nombre de coupes disponibles et c le nombre de capacités disponibles. La complexité de la fonction de calcul vorace est déterminée par la complexité de la fonction de réduction de la solution *reduire*(). Comme sa complexité est de $\theta(n \times c)$, la complexité de calcul vorace est aussi de $\theta(n \times c)$.

La complexité de la fonction d'amélioration locale est plus difficile à déterminer. En effet, sa complexité dépend de la taille de l'ensemble des solutions voisines. Pour calculer une solution voisine, la complexité est à nouveau déterminée par *reduire*(). Donc la complexité pour calculer une solution voisine est de $\theta(n \times c)$. Toutefois, le nombre de solutions voisines possibles est inconnu. L'algorithme d'amélioration locale possède donc une complexité dépendant de la taille de l'ensemble des solutions voisines et de $\theta(n \times c)$.

3. Originalité

Notre méthode est classique et ne présente pas d'originalité particulière. Notons que la recherche de voisinage dans l'algorithme d'amélioration locale force la descente le long de la pente, et ne monte jamais. Cela coupe une partie de l'espace de solution qu'il pourrait sembler intéressant d'explorer. Cependant, nous avons essayé les deux méthodes, et forcer la descente permettait d'obtenir de meilleurs résultats bien plus rapidement.

Nous avons aussi exploré les possibilités d'utiliser une amélioration 2-opt, en échangeant à

chaque fois 2 coupes au lieu d'une. Là aussi, les résultats étaient moins probants, et le temps d'exécution augmentait beaucoup trop (3 fois plus de temps pour trouver une solution équivalente au 1-opt). Il a donc été abandonné.

III. Conclusion

L'algorithme a subi de nombreuses corrections itératives lorsque nous avons pu utiliser le vérificateur de solutions, que nous avons confrontés à nos solutions sur les exemples 2_0, 2_9, 4_8, 5_3, 9_1 et 9_9.

Nos solutions, à l'heure du rendu, ont toutes été validées par ce programme, et nous pensons donc avoir couvert toutes les possibilités.

Que le meilleur gagne !