Sujet 59 : Algorithmes pour la gestion des contraintes en optimisation combinatoire

Matthieu Caron

May 2, 2016

Chapter 1

Introduction

1.1 Présentation du sujet

Un problème d'optimisation combinatoire a pour but de trouver une solution optimisant un ou plusieurs objectifs et répondant à un ensemble de contraintes. Par exemple, le problème du sac-à-dos classique vise à sélectionner un sous-ensemble d'éléments dans une collection de n éléments en maximisant une ou plusieurs fonctions profit et satisfaisant une ou plusieurs contraintes de ressource (la capacité du sac), basée sur des fonctions poids. Une solution est alors spécifiée par une chaîne binaire de taille n, de sorte que chaque variable indique si l'élément correspondant est inclus dans le sous-ensemble des éléments sélectionnés (le sac) ou non.

Contrairement aux approches classiques de la programmation mathématique, les métaheuristiques sont des méthodes de haut niveau à usage général qui sont relativement simples à développer tout étant en mesure de fournir des solutions efficaces en pratique à des problèmes d'optimisation combinatoire difficiles et de grande taille. Lors de la conception de métaheuristiques pour ces problèmes optimisation combinatoire, il existe essentiellement trois catégories générales pour la gestion des contraintes : (1) pénaliser les solutions irréalisables, (2) réparer les solutions irréalisables, ou (3) concevoir une représentation et des opérateurs spécifiques pour le problème à résoudre. Il est bien entendu que la performance d'une technique de gestion de contraintes est fortement liée aux caractéristiques du problème à résoudre. Cependant, savoir quelle approche donnera de meilleures résultats reste une question ouverte.

1.2 Pourquoi avoir pris ce sujet

Tout d'abord car j'ai beaucoup aimé le cours d'Algo de Sophie Tison, ça m'a donné envie de trouver un sujet en rapport avec la résolution de problèmes difficiles. En suite parce que ce sujet allait m'apprendre quelquechose, car jusqu'à présent les algorithmes vus en cours nous apprenez à résoudre des problèmes mono-objectifs. Enfin parce que c'était un projet orienté recherche et qu'il me permettait une bonne autonomie dessus (travail seul, liberté du langage et de l'implémentation)

Chapter 2

le Projet

Le projet s est découpé en plusieurs parties, la première comprendre le problème NP-dur du Flow-Shop de permutation, donc le modéliser et proposer un algorithme efficace pour une fonction d'evalution d'une solution candidate au problème. En second lieu il m'a fallut travailler sur un algorithme d'optimisation de solution pour un problème multi-objectifs soit le Pareto Local Search (PLS) donc encore une fois de l'étudier et de le comprendre et ensuite de l'implémenter pour qu'il corresponde bien à mon model actuel et de faire du bon travail de génie logiciel pour que l'algorithme soit parametrable. Car la troisième partie consiste a générer des résultats pour différents jeux de paramètres pour enfin observer ses résultats et mieux comprendre le problème de Flow-Shop de permutation.

2.1 Flow-Shop de permutation

C'est un problème d'ordonnancement, on possède n taches à effectuer sur m machines et on connait p_{ij} soit le temps que met la tache i sur la machine j et enfin on demande à ce que chaque tache soit fini a un certain moment donné, soit d_i date de fin de la tache i.
Les contraintes :

- toutes les taches doivent s executer une fois sur chaque machine de la machine 1 à la machine m
- l'ordre des taches a effectuer reste le même pour chaque machine
- une machine ne peut traiter qu'une tache a la fois

On appelle ce problème Flow-Shop de permutation car les solutions quandidates peuvent être représentée sous forme d'une permutation soit l'ordre dans lequel on va effectuer les jobs.

Voilà ce que ça donne en Python:

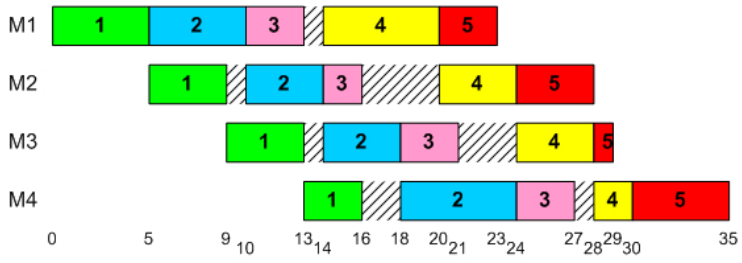
```
class Flowshop(object):
  """
  Le probleme du Flowshop
  Executer toutes les taches sur chaque machine
```

```
    Une tache doit etre terminee pour etre lancee sur une autre machine
    """
    def __init__(self, n, m, p, d, ident=0, type_name="defaut"):
        super(Flowshop, self).__init__()
        # Les donnees d'entree du probleme
        self.type = type_name
        self.id = ident
        self.n = n # nombre de taches INT
        self.m = m # nombre de machines INT
        self.p = p[:] # p[i][j] temps de traitement du job i sur la machine j LIST OF LIST OF
        self.d = d[:] # d[i] date de fin souhaite du job i
        self.cptEval = 0
```

Bien entendu dans un problème NP, la difficulté n'est pas dans la modelisation problème mais surtout dans sa résolution. Mais avant il nous faut faire une fonction d evaluation qui donne un score pour chaque objectif qui ont était étudié durant ce PJI.

- Tmax:Le retard le plus long
- Cmax:La Date de fin du dernier job
- Tsum:Somme des retards
- Usum:Nombre de travaux en retard

Ici voici un exemple de distribution de 5 jobs sur 4 machines.



L'avantage du problème de FlowShop de permutation c est que tous ces objectifs peuvent être calculé lors d'une seule évaluation d'une solution candidate. Et donc voilà ce que ça donne, avec certificat qui correspond a une solution candidate et comme avec PLS on utilise énormément la fonction d'evaluation j'enregistre le resultat dans l'objet certificat pour ne pas le calculer plus d'une fois.

```
def eval(self,certificat,cmax,tsum,tmax,usum) :
    '''
    [FlowshopCertificat]->[bool]->[bool]->[bool]->[bool]->[vecteurScore]
    Entree le certificat a evaluer
          les objectifs a renvoyer si leur valeur vaut True
    Renvoie Cmax,Tsum,Tmax,Usum
    '''
```

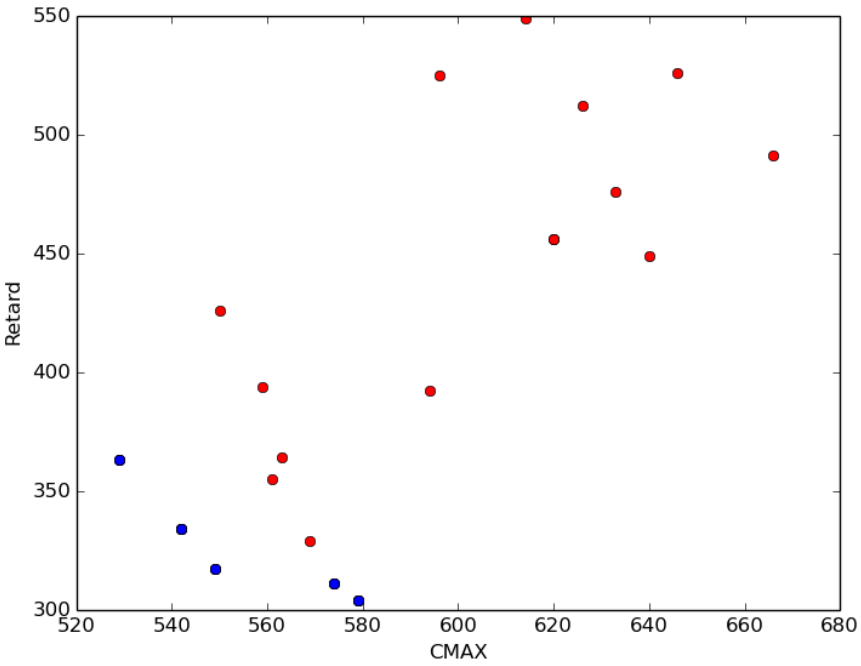
```
if certificat.score != None :
    return certificat.score
travail = [0] * self.m # Tableau temoins du temps de calcul par machine
Tsum = 0 # Somme des retards
Tmax = 0 # Le retard le plus long
Usum = 0 # Nombre de travaux en retard
for iTravail in certificat.permutation :
    for iMachine in range(self.m) :
        if iMachine == 0 : #La machine 0 n'attends personne sauf elle
            travail[0] = travail[0] + self.p[iTravail][0]
        else :
            if travail[iMachine - 1] >= travail[iMachine] :
                travail[iMachine] = travail[iMachine-1] + self.p[iTravail][iMachine]
            else :
                travail[iMachine] = travail[iMachine] + self.p[iTravail][iMachine]
            if iMachine == self.m -1 :
                if travail[iMachine] > self.d[iTravail] :
                    retard = travail[iMachine] - self.d[iTravail]
                    Usum += 1
                    Tsum = Tsum + retard
                    if retard > Tmax :
                        Tmax = retard
Cmax = travail[-1]
vecteurScore = []
if cmax :
    vecteurScore.append(Cmax)
if tsum :
    vecteurScore.append(Tsum)
if tmax :
    vecteurScore.append(Tmax)
if usum :
    vecteurScore.append(Usum)

certificat.score = vecteurScore
self.cptEval += 1
return vecteurScore
```

2.2 Pareto local search

Pour définir ce qu'est l'algorithme de Pareto Local Search (PLS) je vais d'abord définir ce qu'est un optimum de Pareto. C'est lorsqu'il n'existe pas de solution qui "domine" une autre dans un problème a plusieurs objectifs. On dit que a domine b si pour tout objectif j on a $z^j(a) \leq z^j(b)$ avec au moins une infériorité stricte pour un objectif par exemple dans le cas où l'on cherche a minimiser les scores objectifs.

Figure 2.1: Les points bleus forment le front Pareto



Donc le principe de PLS c est de commencer par une solution ou un ensemble de solutions non dominés et de chercher parmis les "voisins" de chaque solution pour progresser petit a petit et ainsi finir avec un ensemble de solutions qui ont des scores objectifs le plus petit possible.

Mon travail n'a pas seulement était d'implémenter l'algorithme de PLS mais aussi de le paramétrer un maximum afin de pouvoir observer le comportement de PLS avec plusieurs de ses variantes. Voici d'abords à quoi ressemble PLS et j'expliquerai les différentes variantes.

Algorithm 1 Pareto Local Search

Require: A_0 un ensemble de solution s non dominées.
  explored(s):= false
  $A := A_0$
  while A_0 n'est pas vide **do**
    s:= solution prise aléatoirement dans A_0
    for all s' **do**
      if s ne domine pas s' **then**
        explored(s') := false
        A := Update(A,s')
      end if
    end for
    explored(s) := true
    $A_0 := s \in A$ quand explored(s)=false
  end while
  RETURN A

2.2.1 Les variantes de PLS

Les voisinnages d'une permutation

Il existe différentes manières d'explorer les différentes permutations, la première choisit la plus simple c est l'exploration contiguë. Par exemple (1,2,3,4) son premier voisin sera (2,1,3,4) son deuxieme voisin (1,3,2,4) son troisieme (1,2,4,3) et enfin (4,2,3,1)

La seconde manière d'explorer les permutations (dans celles que j'ai choisit) c est le swap soit tous les échanges possible entre deux éléments de la permutation.

Et enfin le shift gauche, soit prendre un élément, décallés tous le tableau a gauche et enfin placé l'élément sélectionné en fin du tableau. Exemple :

- (1,2,3,4)
- (1,-,3,4) on selectionne 2
- (1,3,4,-) on décalle a gauche
- (1,3,4,2) on place 2

First ou Best

Par defaut l'algorithme de PLS, tel qu'il est écrit plus haut, représente la version Best, c est à dire qu'on va explorer tous les voisins de la solution selectionnée. Alors que dans la version First, dès qu'on a un voisin qui n'est pas dominé par la solution selectionnée (s) on arrete la boucle et on reselectionne une solution pour laquelle on va explorer son voisinnage. (s) ne sera marqué comme explored que si l'on a visité tous ses voisins, (et si il est toujours dans l'archive A).

Archive

Enfin qu'est-ce qu'on fait de deux evaluations équivalentes pour deux permutations différentes, dans la version classique si une permutation a un même score qu'une autre, elle est ignorée. Dans la variante "archive" on voudrait l'ajouter

et explorer son voisinnage tout en prenant les précautions necessaires pour ne pas boucler.

2.2.2 Résultat

Et donc voilà ce que donne mon implémentation en python 3 de l'algorithme de PLS.

```
def PLS(self,listeVoisins,archive=False,first=True,best=False,trace=False,
                                        cmax=True,tsum=True,tmax=False,usum=False) :
  '''
  Entree : une liste de certificats acceptables
  option trace pour matplotlib
  cf algo du rapport
  '''
  start_time = time.time()
  self.cptEval = 0
  if trace :
       self.doTrace(listeVoisins,'ro',cmax,tsum,tmax,usum)
  for solution in listeVoisins :
       solution.visited = False
  pareto = listeVoisins[:]
  while listeVoisins != []:
    solution = randomPick(listeVoisins)
    while solution.hasNext() :
      solutionPrime = solution.next()
      solutionPrime.visited = False
      scoreSolutionPrime = self.eval(solutionPrime.certificat,cmax,tsum,tmax,usum)
      if trace :
        self.doTrace([solution],'ro',cmax,tsum,tmax,usum)
      dominated = False
      for paretoElement in pareto :
        scorePareto = self.eval(paretoElement.certificat,cmax,tsum,tmax,usum)
        if trace :
               self.doTrace([paretoElement],'ro',cmax,tsum,tmax,usum)
        if self.domine(scoreSolutionPrime,scorePareto) :
               pareto.remove(paretoElement)
        elif self.domine(scorePareto,scoreSolutionPrime) :
               dominated = True
      if not dominated and not solutionPrime in pareto:
        pareto.append(solutionPrime)
        if first==True :
               break
    if not solution.hasNext() :
        solution.visited = True
    listeVoisins = listeNonVisited(pareto)
  listeScore = []
  for voisin in pareto :
       evalFinale = self.eval(voisin.certificat,cmax,tsum,tmax,usum)
       listeScore.append(evalFinale)
```

```
if trace :
      self.doTrace(pareto,'bo',cmax,tsum,tmax,usum)
      plt.show()
timeTotal = time.time() - start_time
return (listeScore,self.cptEval,timeTotal)
```

La variante archive n'a pas encore était réimplémenté car je me suis rendu compte a un moment du PJI que mon ancienne version de PLS était fausse, du coup je l'ai refaite depuis le début.

L'option trace ne marche que si l'on a choisit deux objectifs, cette option me permet surtout de voir si on obtiens bien une belle courbe pareto dans toutes les permutations explorées.

2.3 Génération de résultats et études

Après avoir modélisé le problème de FlowShop, ainsi que toutes les variantes de PLS, le but est d'observer : les scores d'éval des points du front Pareto, la complexité (ici on va compter le nombre de fois où l'on a fait appel à la fonction d'evaluation), et enfin le temps cpu. Je possède 1440 instances differentes du problème de FlowShop, pour chaque probleme je vais commencer avec une permutation aléatoire (soit 30 permutations aléatoires, 30 parce que c est pas trop grand mais suffisant pour faire des statistiques) et appliquer PLS avec toutes ses variantes. En somme la complexité du script a lancé est :

$$1440 * 6 * 30 * 2 * 2 * 3 = 3110400$$

- 1440 le nombre de fichiers (instances)
- 6 le nombre de couples d'objectifs possibles
- 30 le nombre de permutation aléatoire
- 2 soit activer First soit activer Best
- 2 soit activer archive soit ne pas l'activer
- 3 les trois explorations de voisinnages

Lorsque l'utilisation de Python pour modéliser un problème NP-dur est discutable en revanche pour faire un script c est parfait, obtenir la liste de tous les chemins vers les fichiers d'instance se fait en une ligne de code grace au module glob. Et donc le fichier générer en sortie est structuré de la manière suivante :

```
scoreObjectif1 scoreObjectif2
scoreObjectif1 scoreObjectif2
...
#C#
nbEvaluation
#T#
tempsCPU
```

Et donc le script est en cours d exécution et l'étude des résultats obtenus se fait ce mois ci.

Chapter 3

Conclusion

La plus grande compétence que m'a apporté ce PJI en parallèle avec UE de compilation de Pierre Boulet, c est de bien choisir son langage en fonction de l'utilité qu'on va avoir, en effet j'ai une forte utilisation de python et j'arrive a programmer rapidement dessus, gagner du temps Humain c est bien mais quand on se lance dans un projet où l'on va lancer plus de trois millions de recherche de solution sur des problèmes NP-dur il vaut mieux choisir un langage compilé... Enfin c'était intéressant car la plupart du temps les travaux a réaliser on possède une instance du problème et on se base sur sa résolution point. Du coup on peut se permettre de lancer son programme, si ça marche pas on corrige on relance, si ça remarche pas on recorrige et on relance etc... Lorsqu'il s'agit d'écrire un script conséquent, cette méthode ne peut pas s'appliquer. Il faut donc décomposer le code en plusieurs fonctions réduites qui devront être testées.