# Master M1 – optimisation - 2017

# Sujet 2 – Optimisation discrète – utilisation d'heuristiques

L'objectif de ce projet est de vous amener à développer des algorithmes utilisant des heuristiques sur le problème du sac à dos.

# Structuration des classes

Les classes sont structurées de la manière suivantes:

- Le package **`generic`** contient les classes génériques à la base du projet
  - la classe **`AlgorithmeAbstract`** décrit un algorithme
  - la classe abstraite **`Probleme`** décrit le problème à traiter
  - la classe abstraite **`SolutionPartielle`** décrit les solutions en train d'être construite
  - la classe abstraite **`Heuristique`** permet de représenter une heuristique (pour branchandbound et Aetoile)
- le package **`sacADos`** contient l'implémentation du problème du sac à dos
  - la classe **`Objet`** représente un objet du sac à dos ;
  - la classe **`ProblemeSacADos`** modélise un problème ;
  - la classe **`SolutionSacADos`** modélise un sac à dos partiellement rempli ;
  - les classes **`Heuristique`** sont à compléter dans la suite du TP ;
- le package **`algorithme`** contient les classes à écrire ;
- le package par défaut contient différents **`main`** pour lancer vos classes.

## *Probleme*

Le package **`generic`** présente la manière dont est structuré un problème. Un problème est représenté de manière générale par une classe abstraite **`Probleme`** contenant :

- une méthode **`double evaluer(SolutionPartielle s)`** qui permet d'évaluer une solution partielle ;
- une méthode **`SolutionPartielle solutionInitiale()`** qui retourne une solution partielle vide à partir de laquelle partir.

## *SolutionPartielle*

Le package **`generic`** décrit aussi ce qu'est une solution partielle. Une solution partielle est caractérisée par

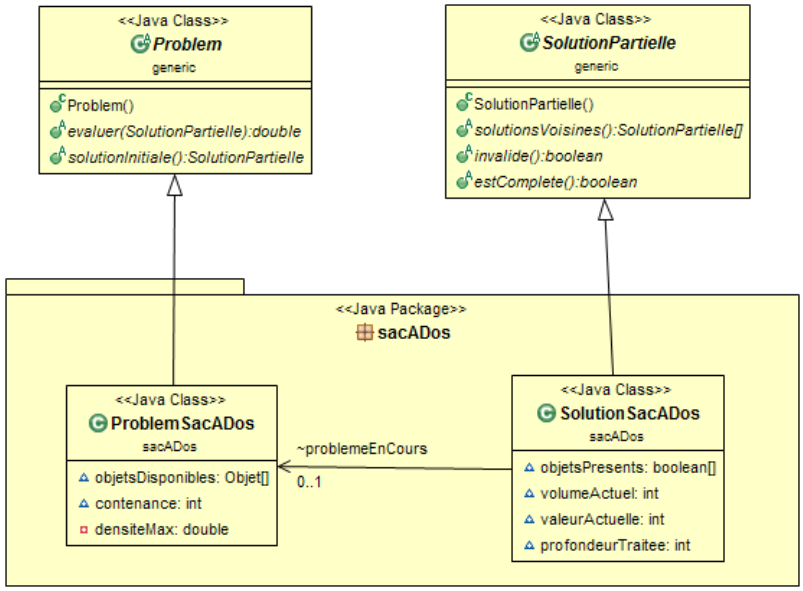
- une méthode **`solutionsVoisines`** qui retourne la liste des solutions qu'il est possible de construire en ajoutant un élément à cette solution partielle. C'est la partie Branch dans un branch-and-bound. L'appel de **`solutionsVoisines`** permet de construire le graphe de construction d'une solution
- une méthode **`estInvalide`** qui permet de savoir quand une solution sort du domaine de

recherche (par exemple le volume contenu dans un sac est trop important)

- une méthode **`estComplete`** qui permet de savoir si une solution partielle est finie (par exemple lorsque tous les objets à mettre dans le sac ont été considérés pour la solution en cours)

### *Problème du sac a dos*

Le problème du sac à dos est modélisé à partir de ces deux classes contenues dans le package **`sacADos`**.




Une solution partielle est définie comme une solution pour laquelle les "**`profondeurTraitee`**" premiers objets ont été affectés. Une solution est **complète** lorsque **`profondeurTraitee`** correspond au nombre d'objets du problème et une solution est **invalide** si le volume des objets ajoutés dans le sac est plus grand que la contenance du sac.

Lorsqu'un dispose d'une **`solutionPartielle`**, les solutions voisines construites correspondent aux deux solutions partielles obtenues lorsqu'on considère l'ajout éventuel de l'objet suivant (ajouté ou non dans le sac).

La solution partielle initiale est une solution dans laquelle aucun objet n'a été considéré (sac vide avant de débuter).

Enfin, certaines méthode **`static`** dans **`ProblemSacADos`** permettent de construire des problèmes (allant du simple au complexe – méthodes **`initialiseProblemeXXX`**).

## Algorithme Generique

La classe **`Algorithme`** du package **`algorithmeAbstract`** du package **`generic`** propose la classe parent des algorithmes par construction. Vos différentes classes algorithme devront hériter de cette classe.

Les différentes classes test dans le package main ont pour objectif de vérifier que votre code fonctionne bien sur des exemples plus ou moins simples

- main.complexe correspond aux main pour un probleme complexe (20 objets)
- main.tresComplexe correspond aux main pour un probleme trés complexe (25 objets)

- main.aleatoire genere à chaque fois un probleme aléatoire (dont la taille est fixée par la constante NB_OBJETS de la classe Constante)

Le descriptif des problemes se trouve dans la classe ProblemSacADos.

Aprés chaque classe, vérifier qu'elle fonctionne avant de passer à la suite.

## Algorithme greedy

Écrire l'algorithme greedy dans la classe **`AlgorithmeGreedy`**.

Il consiste à chaque fois qu'on a le choix à sélectionner la meilleure solution entre deux solutions suivantes (soit on ajoute l'objet soit on décide de ne pas l 'ajouter dans le cas du problème du sac à dos).

Utiliser la variable compteur pour afficher le nombre de nœuds parcourus.

Sur le problème complexe (cf classe **`main.complexe.Test01`**), l'algorithme parcourt 40 nœuds et retourne le sac suivant:

```
40
sac  1 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0(val:45 , vol:49 )
```

## Algorithme parcours en Profondeur

Le parcours en profondeur est donné dans la classe **`AlgorithmeParcoursProfondeur.`**

Exécutez cet algorithme et regardez la réponse fournie. Comparez la à la réponse obtenue via un algorithme greedy.

## Algorithme parcours en largeur

A l'aide d'un algorithme itératif, écrire l'algorithme de recherche complète en largeur d'abord dans la classe **`AlgorithmeParcoursLargeur`**.

Il s'agit de construire à chaque itération k la liste des solutions à la profondeur k, jusqu'à ce que tous les objets aient été considérés (il n'y a plus d'objet à la profondeur suivante). Utiliser la variable compteur pour afficher le nombre de nœuds parcourus.

Sur le problème Complexe (cf **`main.complexe.Test03`**), l'algorithme doit parcourir 2097150 nœuds et retourner le sac suivant:

```
2097150
sac  0 0 0 0 0 1 0 1 0 0 1 1 1 0 1 1 1 0 0 0(val:84 , vol:50 )
```

## Algorithme parcours largeur avec Filtre

Améliorez l'algorithme en n'incluant pas les solutions invalides grâce à la méthode **`doitEtrefiltre`** dans **`AlgorithmeParcoursLargeurFiltre`**

Lorsque le filtre est mis en œuvre (cf main **`Test04`**), l'algorithme parcourt 519924 nœuds et retourne le sac suivant :

```
519924
sac  0 0 0 0 0 1 0 1 0 0 1 1 1 0 1 1 1 0 0 0(val:84 , vol:50 )
```

## Algorithme branch-and-Bound

Écrire l'algorithme branch-and-bound dans la classe **`AlgorithmeBranchAndBound`**

Un algorithme branch-and-bound utilise une heuristique pour essayer d'élaguer des branches. Faites un parcours en largeur et supprimer les branches non pertinentes à l'aide d'une heuristique. Utiliser la variable compteur pour afficher le nombre de noeuds parcourus.

En utilisant l'heuristique de base **`HeuristiqueDensiteMax`** (cf main **`Test05`**) , une approche branch-and-bound appliquée au problème complexe fournira les résultats suivants :

```
336168
sac  0 0 0 0 0 1 0 1 0 0 1 1 1 0 1 1 1 0 0 0(val:84 , vol:50 )
```

## Heuristiques

Dans les tests Branch and bound trié, le problème est d'abord trié par densité d'objets. Remarquez les différences en terme de performance.

Lorsqu'on trie les éléments et qu'on utilise des heuristiques plus complexes, il est possible de faire beaucoup mieux

Avec une heuristique simple consistant à considérer la densité maximale des objets restants, on obtient sur un problème complexe (test 07).

```
992
sac  1 1 1 1 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0(val:84 , vol:50 )
```

et avec une heuristique plus réaliste en regardant précisément le nombre d'objets restants et en les triant par densité (relaxation du problème entier --> reel), on arrive

```
918
sac  1 1 1 1 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0(val:84 , vol:50 )
```

Expliquez les différentes heuristiques et les raisons des différences de performances dans les tests branch-and-bound (test 05-08).

## Algorithme A*

Écrire l'algorithme A* dans la classe **`AlgorithmeAEtoile`**.

Désormais, au lieu de faire une recherche en largeur, l'algorithme A* ne va développer que la solution partielle la plus prometteuse (celle pour laquelle évaluation parcourue + estimation est la plus faible).

**A l'aide d'une Liste triée** (ou d'une queue de priorité en java), sélectionner à chaque pas de temps le meilleur élément et développer le en ajoutant les nouvelles solutions obtenues dans cette liste de priorité.

L'utilisation A* dans le cadre d'un problème dont les objets sont triés par densité avec une heuristique **`Realiste`** parvient à trouver la solution optimale au problème complexe en se limitant à 62 nœuds.

```
62
sac  1 1 1 1 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0(val:84 , vol:50 )
```

## Représentation commune des algorithmes

A l'aide de la classe **`algorithmeFormat.AlgorithmeFormat`**, réécrivez les algorithmes en respectant la structure proposée par le cours.

Cette structure permettra de mettre en avant les variations entre les algorithmes.

## Perspective

En fonction du temps qui vous reste

- Vous piuvez écrire une heuristique réaliste qui ne nécessite pas d'avoir les objets triés. La difficulté est principalement algorithmique. Attention, si l'heuristique prend beaucoup de temps à calculer le gain dans la recherche peut etre compensé par le temps des estimations des heuristiques.
- Vous pouvez programmer un autre probleme d'optimisation combinatoire (par exemple le TSP)
- Vous pouvez essayer de structurer l'ensemble des algorithmes vu précédemment grace à **`algorithmeFormat.AlgorithmeFormat`**
- Vous pouvez utiliser une classe **`Factory`** pour générer les algos souhaités associés à un problème et faire automatiquement des batteries de test.
- Enfin, le véritable exercice permettant de vérifier que vous avez bien compris l'ensemble serait de prendre un problème d'optimisation combinatoire (par ex, le probleme du sac à dos, le TSP, affectation de tache...), et de tenter de le résoudre au mieux à partir de rien (formaliser le problème, écrire les algorithmes, construire des heuristiques, ...).

## En conclusion

En utilisant des approches A* et une heuristique adaptée, on arrive à réduire d'un facteur 200 par rapport à un branch-and-bound et d'un facteur de 1 million par rapport à une recherche brutale.

Bien évidemment, il s'agit d'une expérience sur un exemple donné, et il faudrait faire de nombreux autres expériences, mais ces premiers résultats vous donnent une idée de la force de ces algorithmes.