Université de Nantes — UFR Sciences et Techniques
Master Informatique parcours “Optimisation en Recherche Opérationnelle (ORO)”
Année académique 2021-2022

Rapport du Devoir Maison n°1

Métaheuristiques

Nicolas COMPÈRE – Adrien CALLICO

September 28, 2021

Devoir maison 1 :
Heuristiques de construction et d'amélioration gloutonnes

Formulation du SPP

Le SPP (Set Packing Problem) est un problème d'optimisation combinatoire NP-complet. Il est défini par le modèle mathématique suivant :

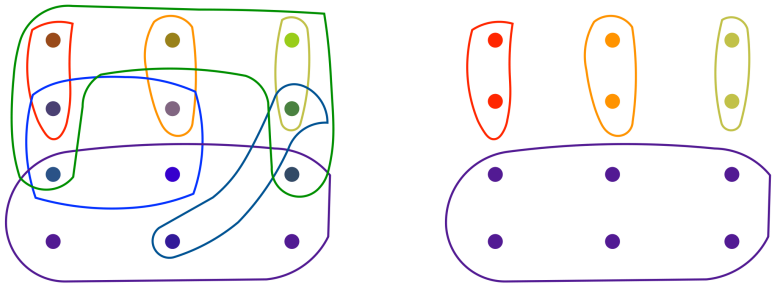
$$
\begin{aligned}
max \quad & \sum_{i=1}^{n} c_i x_i \\
s.c. \quad & \sum_{i=1}^{n} t_{ij} x_i \leq 1 \quad \forall j \in \{1, .., m\} \\
& x_i \in \{0, 1\} \quad \forall i \in \{1, .., n\}
\end{aligned}
$$

Une application concrète de ce problème est la situation suivante :
Un vendeur possède n différents objets à vendre. Il y a m acheteurs, qui veulent chacun acheter un sous-ensemble de ces objets. On attribue à chaque sous ensemble un certain prix.

Il faut alors du sous-ensemble d'acheteurs auquel le vendeur attribuera les sous-ensembles des objets qu'ils veulent acheter, sachant qu'on ne peut vendre un même objet à deux acheteurs différents, et qu'on souhaite bien évidemment maximiser le montant de la vente.

Une illustration :

Les points représentent les objets, et les sous-ensembles les objets qui intéressent les acheteurs. On veut alors déterminer un ensemble de sous-ensembles disjoints, en maximisant le montant de la vente (on n'est pas obligé de tout recouvrir)



(a) Situation au départ

(b) Situation à l'issue de la résolution

Figure 1: SPP Illustré [1]

[1] `https://algorist.com/problems/Set_Packing.html`

Modélisation JuMP (ou GMP) du SPP

Nous utilisons le solver GLPK

```
model = Model(GLPK.Optimizer)
```

(optionnel) Nous avons défini une limite de durée de 300 secondes

```
set_time_limit_sec(model, 300.0)
```

Les variables binaires représentent la décision de sélectionner (ou pas) un acheteur.

```
@variable(model, x[1:n], Bin)
```

L'objectif du vendeur est de maximiser le montant de la vente.

```
@objective(model, Max, dot(C, x))
```

Le vendeur ne possède qu'un exemplaire de chaque type d'objet : il ne peut donc pas vendre le même type d'objet à deux acheteurs différents.

```
 @constraint(model , cte[i=1:m], sum(A[i,j] * x[j] for j=1:n) <= 1)
```

Instances numériques de SPP

Instance	# Variables	# Contraintes	Densité (%)	Meilleure valeur connue
100rnd0100	100	500	2.0	372
100rnd0300	100	500	2.96	203
1000rnd0300	1000	5000	0.6	661
1000rnd0700	1000	1000	0.58	2260
200rnd1300	200	600	1.5	571
500rnd1100	500	500	2.26	424
500rnd0500	500	2500	2.22	122
2000rnd0600	2000	2000	2.56	9
200rnd0700	200	200	1.53	1004
200rnd0500	200	1000	2.48	184

Heuristique de construction appliquée au SPP

L'algorithme de construction reprend étape par étape l'algorithme de construction glouton présenté dans le chapitre 2 du cours :

1. On initialise le vecteur solution avec des zéros (aucun acheteur encore sélectionné) et celui des sous ensembles candidats avec des uns (ils sont tous disponibles)

2. Tant qu'il reste des sous ensembles disponibles :
    - on calcule l'utilité (détaillée ci-dessous) de chaque acheteur et on sélectionne le plus intéressant.
    - on l'ajoute à notre solution
    - on détermine quels acheteurs sont en conflit avec l'acheteur que l'on vient de sélectionner (ils veulent un objet commun) et on ne les prendra plus en compte pour les prochaines itérations.
3. On retourne le vecteur solution

Notre première idée pour calculer l'utilité de chaque candidat était de simplement prendre en compte la valeur de chaque sous-ensemble et de sélectionner celui qui a la plus grande.

Cette méthode ne donnait pas de bon résultats, et nous nous sommes alors inspirés de l'exercice de TD sur le problème du sac-à-dos unidimensionnel en variables binaires, en calculant le ratio

$$\frac{\text{valeur associée à la variable}}{\text{son nombre d'apparition dans les contraintes}}$$

pour chaque variable. Le fait de sélectionner le ratio le plus grand permet de choisir en priorité les sous-ensembles avec une valeur importante et/ou apparaissant peu dans les contraintes (peu disputé par les acheteurs).

Concernant les choix d'implémentations, nous avons décidé, plutôt que de coûteuses modifications sur la matrice t (comme par exemple remplacer des coefficients, voire supprimer des lignes/colonnes) de passer par des vecteurs de variables binaires qui représentent si une variable ou une contrainte a déjà été prise en compte. Ainsi, on ne modifie alors pas du tout la matrice pour calculer les nouvelles évaluations à chaque itération.

En revanche, on calcule l'ensemble des évaluations à chaque fois, et on met ensuite à 0 les évaluations des variables déjà utilisées ou impossible à sélectionner. Une amélioration possible serait de calculer uniquement les évaluations utiles.

Exemple : l'instance didactic.dat

On initialise le vecteur solution avec des zéros, puis on calcule l'évaluation de chaque variable :

évaluations : [3.3333333333333335, 1.25, 2.0, 3.0, 2.25, 6.5, 5.5, 1.0, 1.5]

On sélectionne alors la variable 6 : on l'intègre à la solution ($x_6 = 1$), puis on élimine les variables qui feront conflit :

Variables disponibles après ajout : [0, 0, 0, 1, 0, 0, 1, 0, 0]

On calcule de nouveau les évaluations en prenant en compte les variables restantes :

évaluations : [0.0, 0.0, 0.0, 3.0, 0.0, 0.0, 5.5, 0.0, 0.0]

On sélectionne la 7 :

variables disponibles après ajout [0, 0, 0, 1, 0, 0, 0, 0, 0]

On recalcule les évaluations : [0.0, 0.0, 0.0, 3.0, 0.0, 0.0, 0.0, 0.0, 0.0]

On sélectionne la 4

Variables disponibles après ajout [0, 0, 0, 0, 0, 0, 0, 0, 0]

Il n'y a plus de variables disponibles, l'algorithme s'arrête et renvoie la solution
$[0, 0, 0, 1, 0, 1, 1, 0, 0]$

Heuristique d'amélioration appliquée au SPP

Au vu des relativement bons résultats de l'algorithme de construction glouton[2], nous avons d'abord implémenté une heuristique d'amélioration par recherche locale en plus profonde descente, car c'est la méthode qui nous semblait être la plus à même d'améliorer la solution construite par l'algorithme glouton.

Cependant, en modifiant légèrement nos algorithmes pour passer d'une plus profonde descente à une simple descente, il s'est avéré que la simple descente donnait des résultats aussi bons voire meilleurs, en bien moins de temps, surtout si l'instance est de grande taille.

Nous avons opté pour trois voisinages de type kp-exchange : 2-1 exchange, 1-1 exchange et 0-1 exchange, que nous regardons en totalité à chaque fois.

Nos algorithmes de simple et profonde descente les parcourent dans cet ordre.

Concernant les implémentations des algorithmes de recherche de solutions aux voisinges précédemment cités, elles sont assez naïves (on évite quelques doublons mais pas tous) : sans pré-traitement pour sélectionner quelles sont les variables les plus intéressantes à échanger, la complexité pour le 21-exchange est en effet en $O(n^3)$. (on parcourt environ trois fois la liste des variables)

C'est une des pistes d'amélioration de nos heuristiques. (voir Discussion)

Ce dernier point nous a décidé à choisir la descente simple plutôt que la plus profonde descente.

Exemple : l'instance didactic.dat

Pour cet exemple, nous avons choisi d'appliquer la simple descente. On part d'une solution admissible pour l'instance didactic.data : $x_0 = [0, 0, 1, 1, 0, 0, 0, 0, 0]$, avec $z = 14$

L'algorithme va d'abord déterminer si un 21-exchange améliorant est possible, et n'en trouve pas.

Il réessaye avec le 11-exchange, et trouvera que passer x_4 à 1 et x_2 à 0 augmentera la valeur de la fonction objectif (de 1...). Il arrêtera donc d'essayer d'autres 11-exchanges (car nous sommes en descente simple).

Il ne trouvera pas non plus de 01-exchange améliorant.

Il retournera donc la solution $x_0 = [0, 1, 1, 0, 0, 0, 0, 0, 0]$ avec $z = 15$.

Expérimentation numérique *(graphiques en annexe)*

Nos algorithmes ont été implémentés en Julia (version 1.6.0)

L'environnement machine dans lequel nous avons réalisé nos tests présente les caractéristiques suivantes :

Système d'exploitation Ubuntu 20.04

Processeur Intel Core™ i5-7300HQ CPU @ 2.50GHz × 4

Carte Graphique Mesa Intel HD Graphics 630 (KBL GT2)

RAM 8 Go

[2] voir les sections Expérimentation numérique et Discussion

Glouton :

Instance	z	Temps (s)
pb_100rnd0100	342	0.014
pb_100rnd0300	193	0.003
pb_1000rnd0300	507	0.878
pb_1000rnd0700	2050	0.543
pb_200rnd1300	497	0.013
pb_500rnd1100	344	0.036
pb_500rnd0500	84	0.032
pb_2000rnd0600	7	0.165
pb_200rnd0700	945	0.010
pb_200rnd0500	162	0.005

Recherche locale en plus profonde descente:

Instance	zInit	z	Temps (s)	Amélioration par rapport à zInit (%)
pb_100rnd0100	342	350	0.077	2.3
pb_100rnd0300	193	194	0.019	0.5
pb_1000rnd0300	507	520	26.221	2.5
pb_1000rnd0700	2050	2057	91.613	0.34
pb_200rnd1300	497	507	0.535	1.97
pb_500rnd1100	344	350	1.019	1.71
pb_500rnd0500	84	84	0.552	0
pb_2000rnd0600	7	7	9.960	0
pb_200rnd0700	945	956	0.541	1.15
pb_200rnd0500	162	162	0.069	0

Recherche locale en simple descente:

Instance	zInit	z	Temps (s)	Amélioration par rapport à zInit (%)
pb_100rnd0100	342	350	0.074	2.3
pb_100rnd0300	193	194	0.049	0.5
pb_1000rnd0300	507	513	13.846	1.17
pb_1000rnd0700	2050	2069	4.181	0.9
pb_200rnd1300	497	510	0.063	2.5
pb_500rnd1100	344	355	0.259	3
pb_500rnd0500	84	84	0.673	0
pb_2000rnd0600	7	7	8.795	0
pb_200rnd0700	945	956	0.044	1.1
pb_200rnd0500	162	162	0.079	0

Résultats du solver GLPK (Nous avons mis une limite de temps de 300s) :

Instance	z	Temps (s)
pb_100rnd0100	372	1.684
pb_100rnd0300	203	0.685
pb_1000rnd0300	471	302.794
pb_1000rnd0700	2219	300.250
pb_200rnd1300	571	65.574
pb_500rnd1100	409	300.0722
pb_500rnd0500	122	300.444
pb_2000rnd0600	5	300.169
pb_200rnd0700	1004	0.006
pb_200rnd0500	184	42.257

Remarque commune à chaque expérimentation : à l'aide des graphiques situés en annexe, on observe qu'un autre facteur prépondérant à l'augmentation de la durée des expérimentations (avec la taille des instances) est la densité de la matrice.

Discussion

Nous avons tenté de résoudre à l'optimalité diverses instances à l'aide du solver GLPK. Pour certaines, nous avons arrêté la résolution en cours de route, voyant que plusieurs dizaines de minutes n'étaient pas suffisantes.

Ainsi, nous avons décidé de présenter les résultats de GLPK sur 10 instances sélectionnées avec un limite de temps de 300s.

L'usage des heuristiques apparaît alors comme intéressant pour les instances de grande taille : elles fournissent une solution de bonne facture en quelques secondes au pire (pour la descente simple).

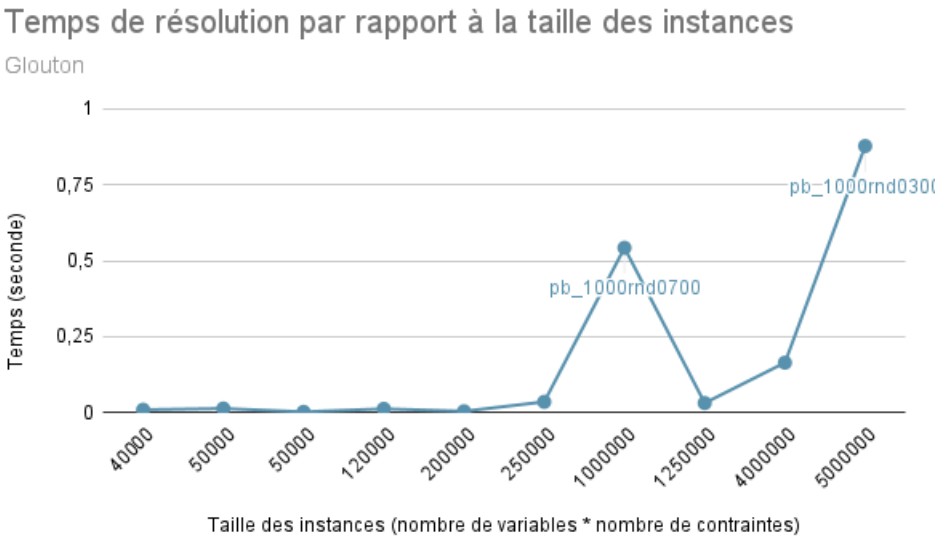
Pour les instances que le solver a réussi à résoudre à l'optimalité, l'heuristique de construction fournit une solution avoisinant 90% de la solution optimale, pour un temps toujours inférieur à 1 seconde, infime comparé au temps requis par le solveur pour certaines instances.

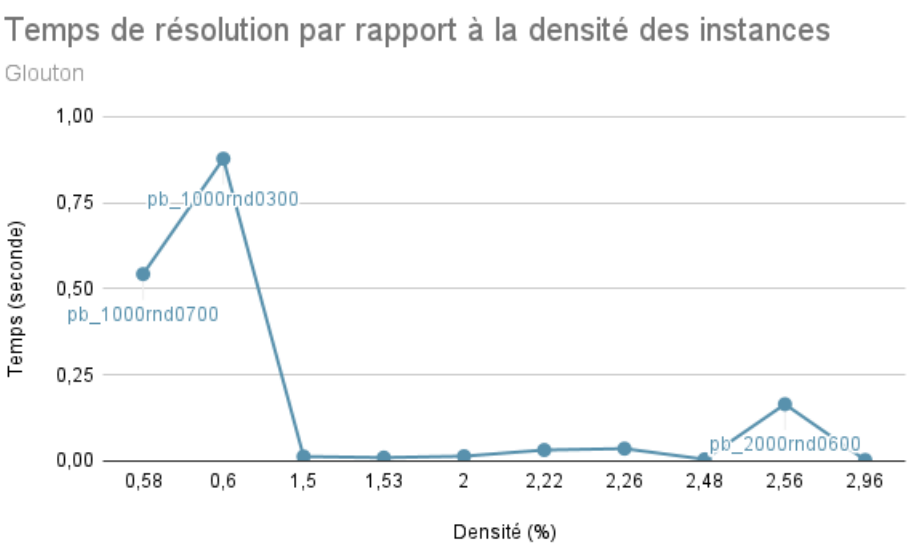
L'emploi de (méta)heuristiques est ainsi prometteur, avec un soupçon de réserve toutefois :

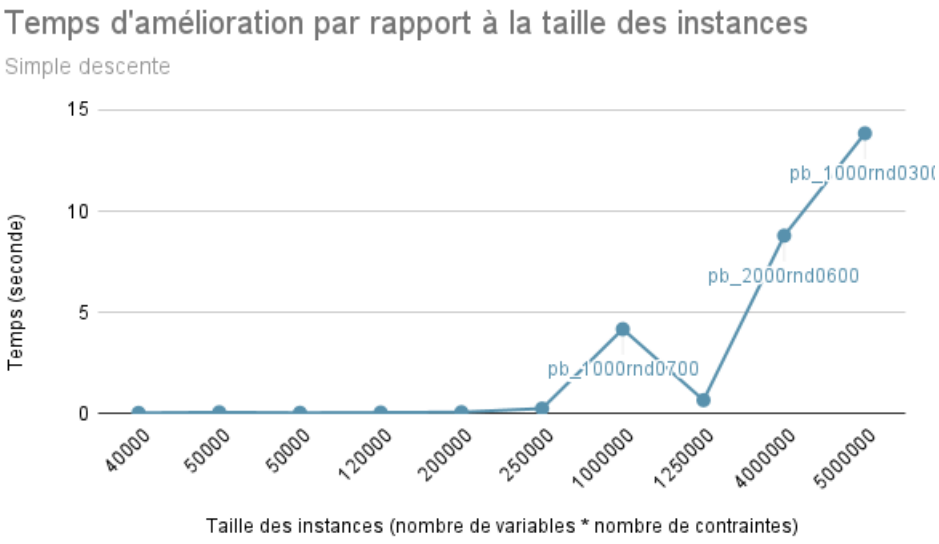
- Il faut accepter de faire un compromis sur l'optimalité de la solution (pour la plupart des instances) en échange d'un temps de calcul réduit.
- Un pré-traitement ou du moins une étude des données semble nécessaire : on pourrait par exemple utiliser des structures de données plus adaptées lorsque la matrice t a une faible densité.
- Sur plusieurs instances, l'heuristique d'amélioration n'arrivait pas à améliorer la solution construite par l'algorithme glouton. Une solution serait d'en utiliser d'autres, qui ne se feraient pas piéger trop facilement dans un maximum (minimum) local, tout en gardant une complexité raisonnable.

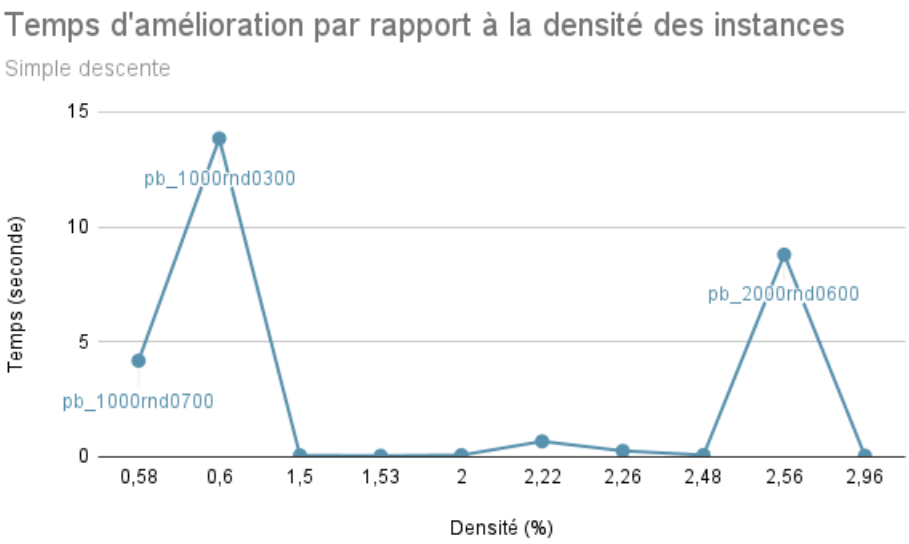
Les améliorations que nous envisageons pour nos heuristiques sont :

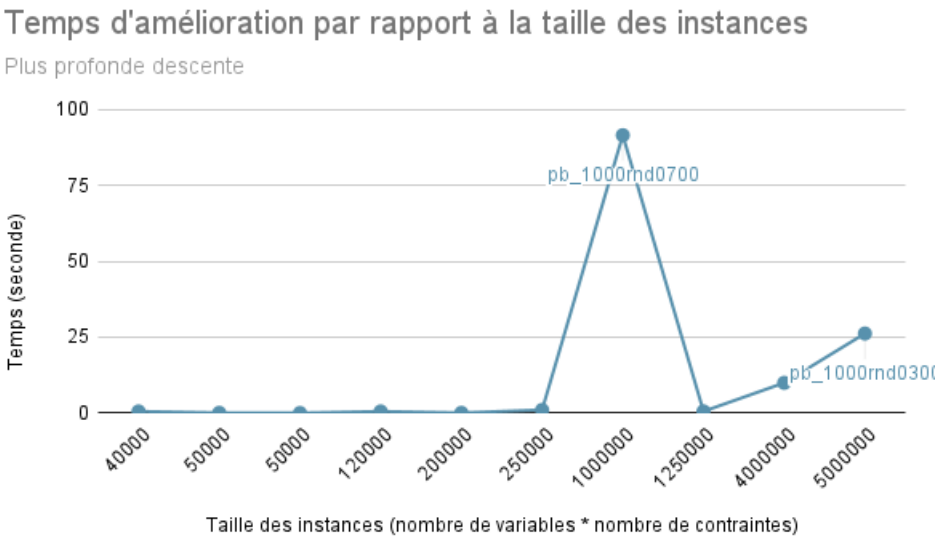
- Réduire le nombre d'évaluations calculées lors de l'algorithme glouton
- Utiliser d'une structure de donnée plus adaptée que la matrice, car celle-ci contient beaucoup d'informations "inutiles" (on doit parcourir plusieurs fois toute la matrice qui est potentiellement de grande taille, alors qu'on a uniquement besoin de savoir l'emplacement des coefficients égaux à 1, pour savoir à quelle(s) contrainte(s) est(sont) liée(s) chaque variable)
- Déterminer pour chaque kp-exchange quels sont ceux qui répondent aux contraintes, puis de vérifier si c'est une solution améliorante (dans les algorithmes que nous avons implémentés, nous envisageons toutes les possibilités (parfois en doublon lors du 21-exchange) Voire essayer de déterminer (à l'aide des coefficients de la fonction objectif) ceux qui ont le plus de chances d'être une solution améliorante (un peu comme les ratios calculés pendant l'algorithme glouton).


Temps de résolution par rapport à la taille des instances
Glouton
1
0,75
0,5
0,25
0
Temps (seconde)
pb_1000rnd030(
pb_1000rnd0700
40000
50000
50000
120000
200000
250000
1000000
1250000
4000000
5000000
Taille des instances (nombre de variables * nombre de contraintes)


Temps de résolution par rapport à la densité des instances
Glouton
1,00
0,75
0,50
0,25
0,00
Temps (seconde)
pb_1000rnd0300
pb_1000rnd0700
pb_2000rnd0600
0,58
0,6
1,5
1,53
2
2,22
2,26
2,48
2,56
2,96
Densité (%)

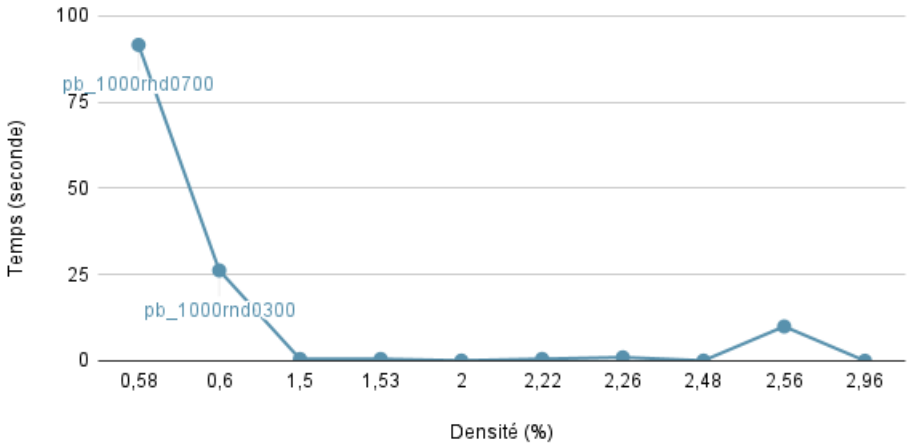

Temps d'amélioration par rapport à la taille des instances
Simple descente
15
10
5
0
Temps (seconde)
pb_1000rnd0700
pb_2000rnd0600
40000
50000
50000
120000
200000
250000
1000000
1250000
4000000
5000000
Taille des instances (nombre de variables * nombre de contraintes)


Temps d'amélioration par rapport à la densité des instances
Simple descente
15
10
5
0
Temps (seconde)
pb_1000rnd0300
pb_2000rnd0600
pb_1000rnd0700
0,58
0,6
1,5
1,53
2
2,22
2,26
2,48
2,56
2,96
Densité (%)


Temps d'amélioration par rapport à la taille des instances
Plus profonde descente
100
75
50
25
0
Temps (seconde)
pb_1000rnd0700
40000
50000
50000
120000
200000
250000
1000000
1250000
4000000
5000000
Taille des instances (nombre de variables * nombre de contraintes)

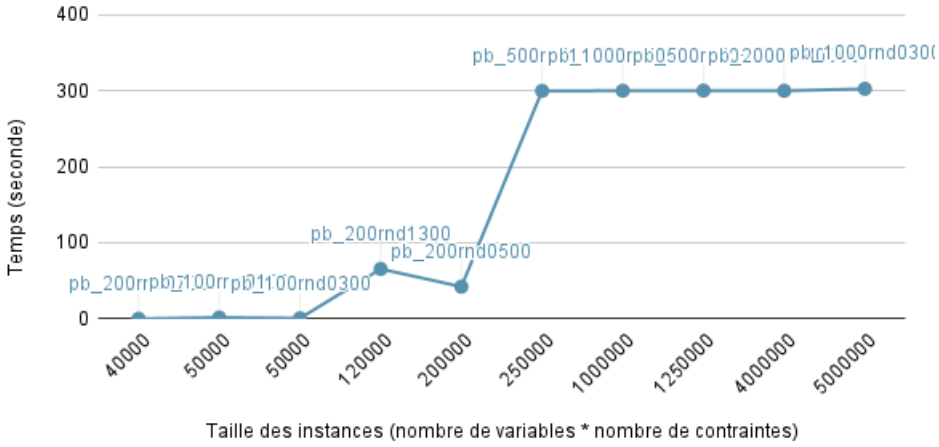
Temps d'amélioration par rapport à la densité des instances

Plus profonde descente



Temps de résolution par rapport à la taille des instances

Résolution optimale



Temps de résolution par rapport à la densité des instances

Résolution optimale