Compte Rendu : Optimisation Combinatoire

Benjamin DANGLOT

3 janvier 2016

1 Mono-objectif

1.1 Implémentation

Dans cette section, on définit l'implémentation des différentes composantes des algorithmes pour tenter de résoudre les instances de SMTWTP :

1.1.1 Sélection de solution

FirstImprovement(*First*)	Selection de la première solution améliorante
BestImprovement(*Best*)	Génération entière du voisinage, puis selection de la meilleure solution voisine.

1.1.2 Heuristiques de première Solution

Random(*rnd*)	Génération d'une solution Aléatoire
Earliest Due Date(*Edd*)	Génération d'une solution en fonction de la DeadLine par ordre croissant
Modified Due Date(*Mdd*)	Génération d'une solution avec la somme des tâches déjà ordonnancées

1.1.3 Génération de Voisinage

insert	insérer un job position i à la position j
exchange	échange de position de job successifs
swap	échange de position de deux jobs i et j

1.1.4 Données

On travaillera sur le fichier *wt100.txt*, qui contient 125 instances de 100 jobs du problème SMTWTP.

1.1.5 Difficultés

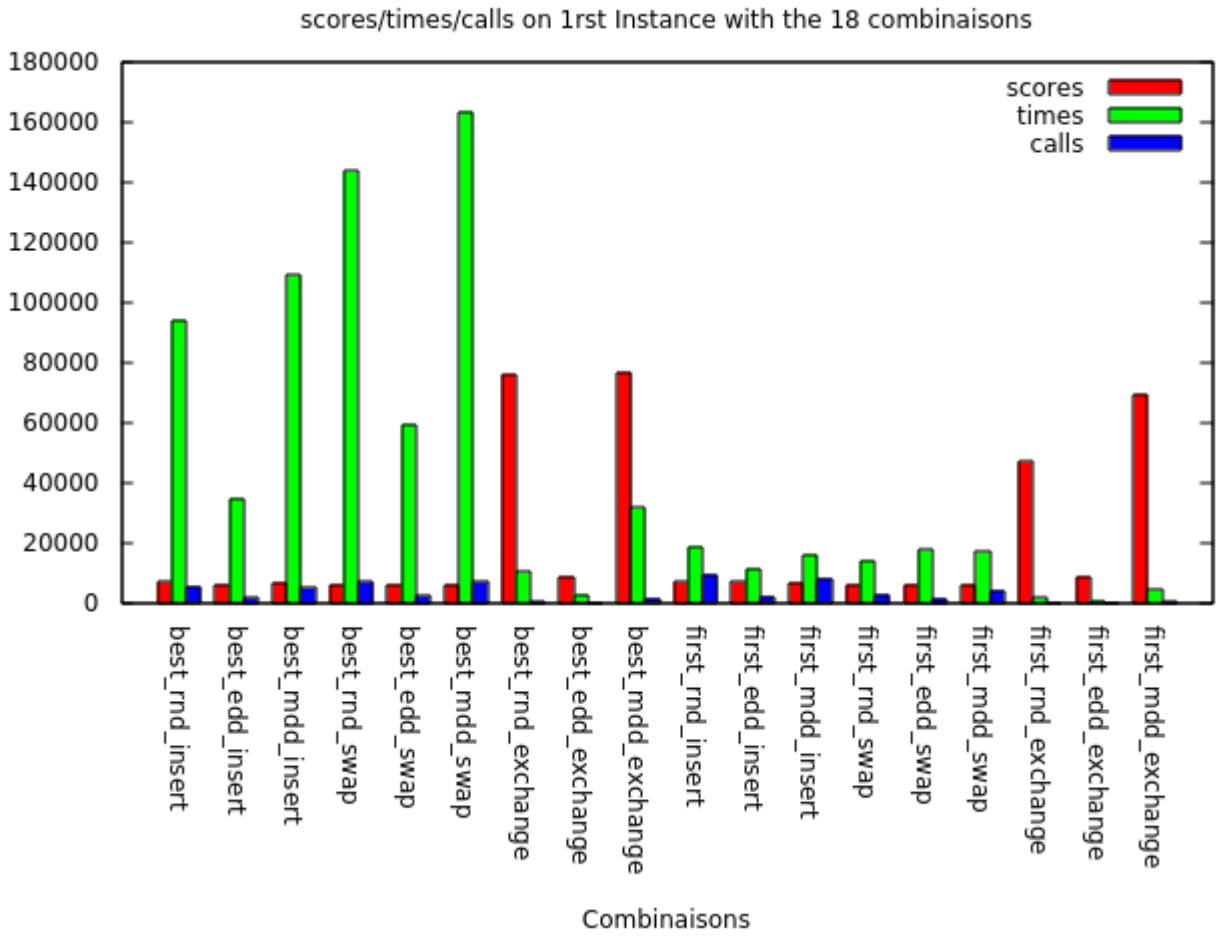
Au cours de mes expériences, j'ai pu constater que les instances n'ont pas la même d̈ifficultés.̈ Dans une optique de clarté des prochains graphiques, j'ai décidé de classé les instances suivants 4 n̈iveauẗde difficultés suivant le score :

- ≤ 10000 difficulté 1
- $10000 <$ score ≤ 100000 difficulté 2
- $100000 <$ score ≤ 500000 difficulté 3
- > 500000 difficulté 4

1.2 Recherche Locale

1.2.1 Premiers Résultats

J'ai choisi d'effectuer une première mesure sur la première instance. Voici les histogrammes des scores, du temps et du nombre d'appel à la fonction de fitness par les 18 configurations possibles (Sélection(2) * Heuristiques(3) * Voisinage(3) = 18).



(a) Score, Temps et nombre d'appel a la fonction de fitness pour la première instance avec les 18 combinaisons

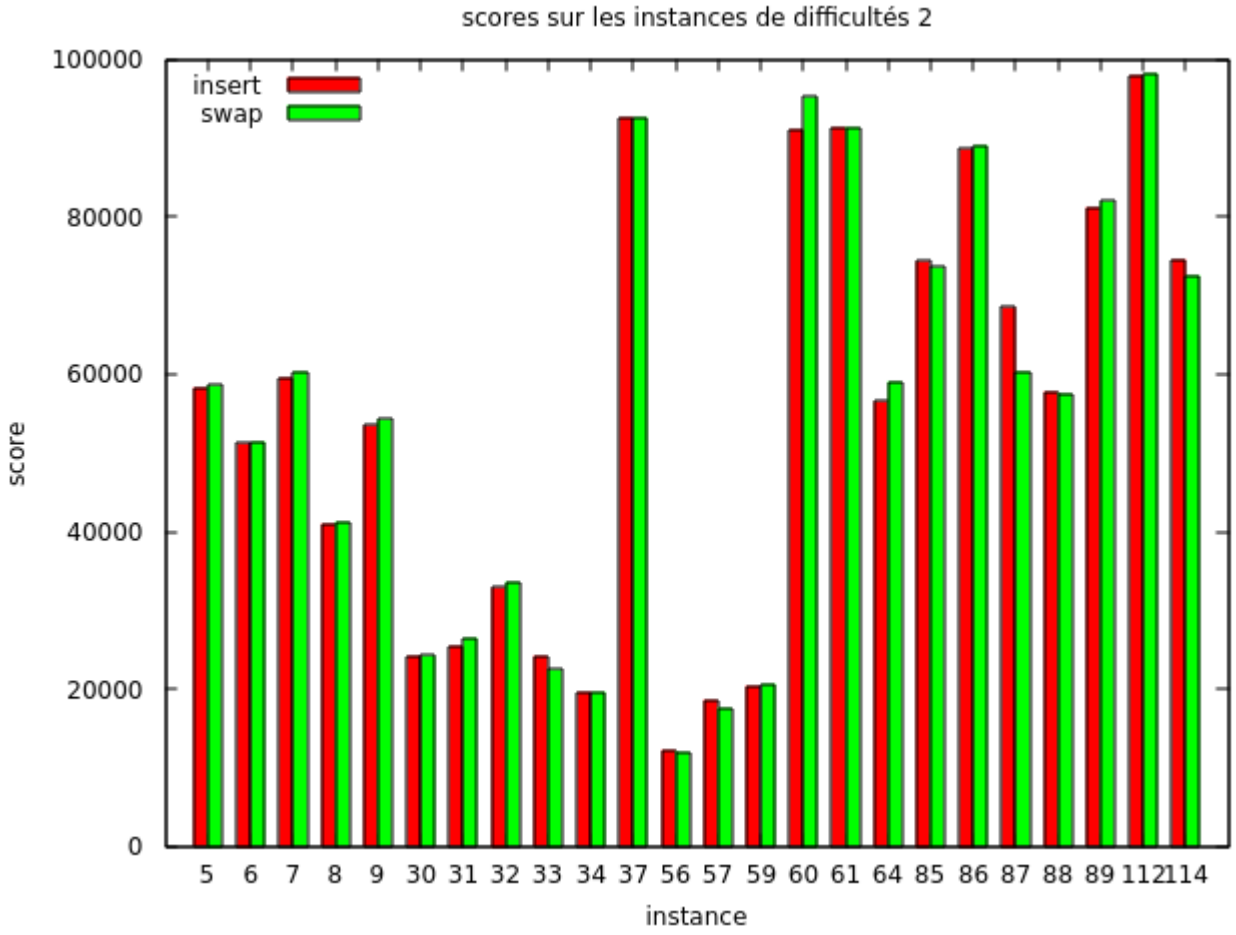
On ne peut pas tirer grand chose de ce graphiques, à part de confirmer les a priori : les heuristiques *rnd* convergent rapidement dans un minimum local (peu d'appel à la fonction de fitness) et le voisinage *exchange* ne donne pas de bon résultat en terme de score et est plus petit que les deux autres. On peut également dire, que la sélection *best* consomme plus de temps, mais donne des résultats un peu mieux.

1.2.2 Comparaison de deux voisinage

J'ai sélectionné deux combinaisons dont la différences est uniquement la génération des voisins. Nous allons étudier *best-edd* avec les deux voisinages *insert* et *swap* car ces combinaisons on atteint l'optimum connus pour la première instance :

Voisinage	Score	Temps	Appel
insert	5988	1644	34659
swap	5988	2539	59414

Voici les scores obtenus avec *best-edd-insert* en rouge et *best-edd-swap* en vert sur les instances de difficultés 2 :



(a) Score pour *best-edd-insert* en rouge et *best-edd-swap* en vert sur les instance de difficultés 2

En résumé, le voisinage *insert* b̈atl̈e voisinage *swap* sur 12 instances, et il y a 4 ex aequo. (*swap* gagne donc sur 8 instances)

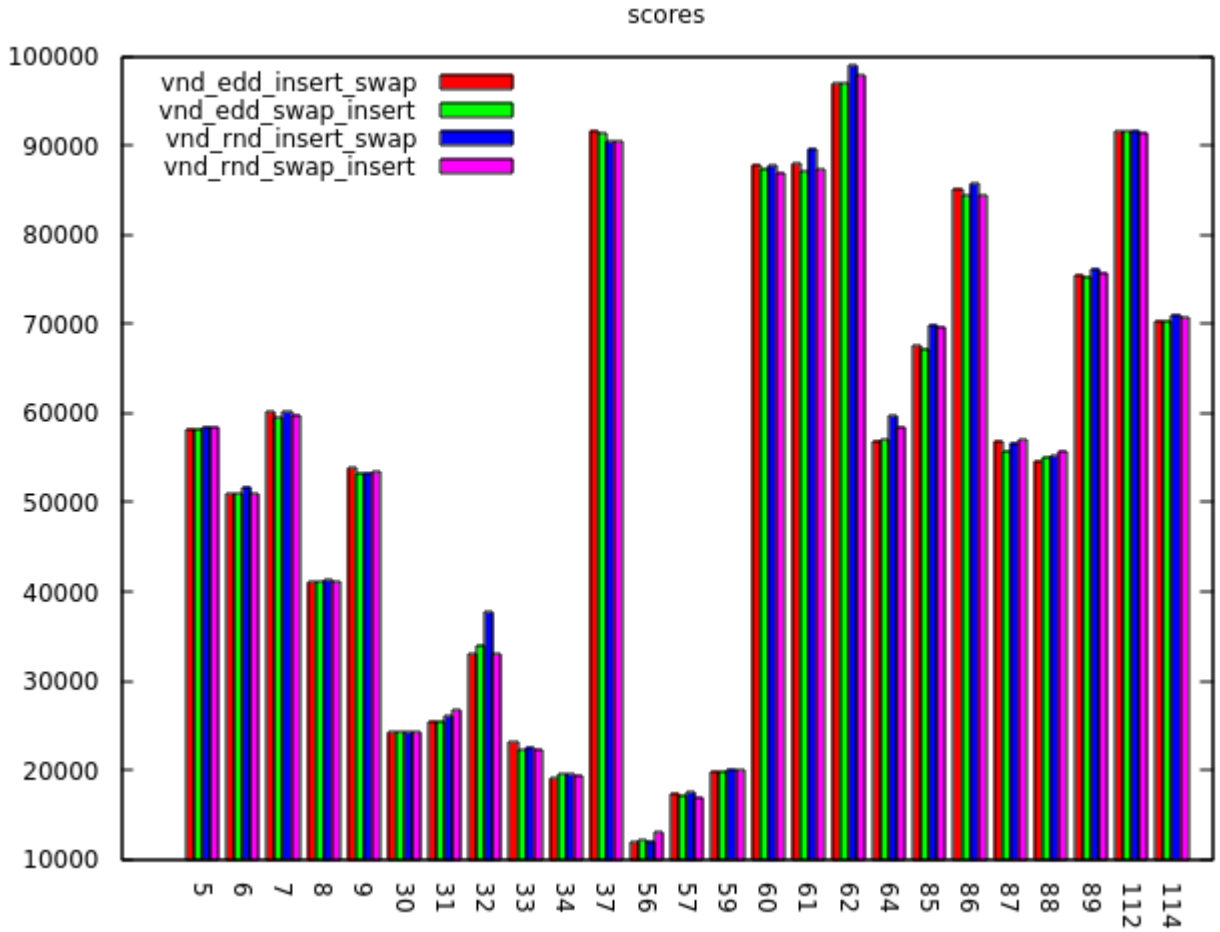
1.3 Variable Neighborhood Descent

Dans cette section, nous étudierons 4 algorithmes avec 2 variantes d'exécutions chacune détaillée dans les sous-section 1.3.1 et 1.3.2. Les algorithmes sont construits avec deux listes de voisinages, et deux heuristiques de départ :

- Voisinages
  - *exchange*, *swap*, *insert*
  - *exchange*, *insert*, *swap*
- Heuristiques
  - *Random*
  - *Earliest Due Date*

1.3.1 Variable Neighborhood Descent (VND)

La Variable Neighborhood Descent (VND) procède de la façon suivante : il itère sur différents voisinage, s'il ne trouve pas de solution améliorante, il passe au voisinage suivant. Une fois qu'il a trouvé une solution meilleure que la solution courante, on retourne au premier voisinage (dans notre cas exchange). Voici le graphiques des instances de difficultés 2 :



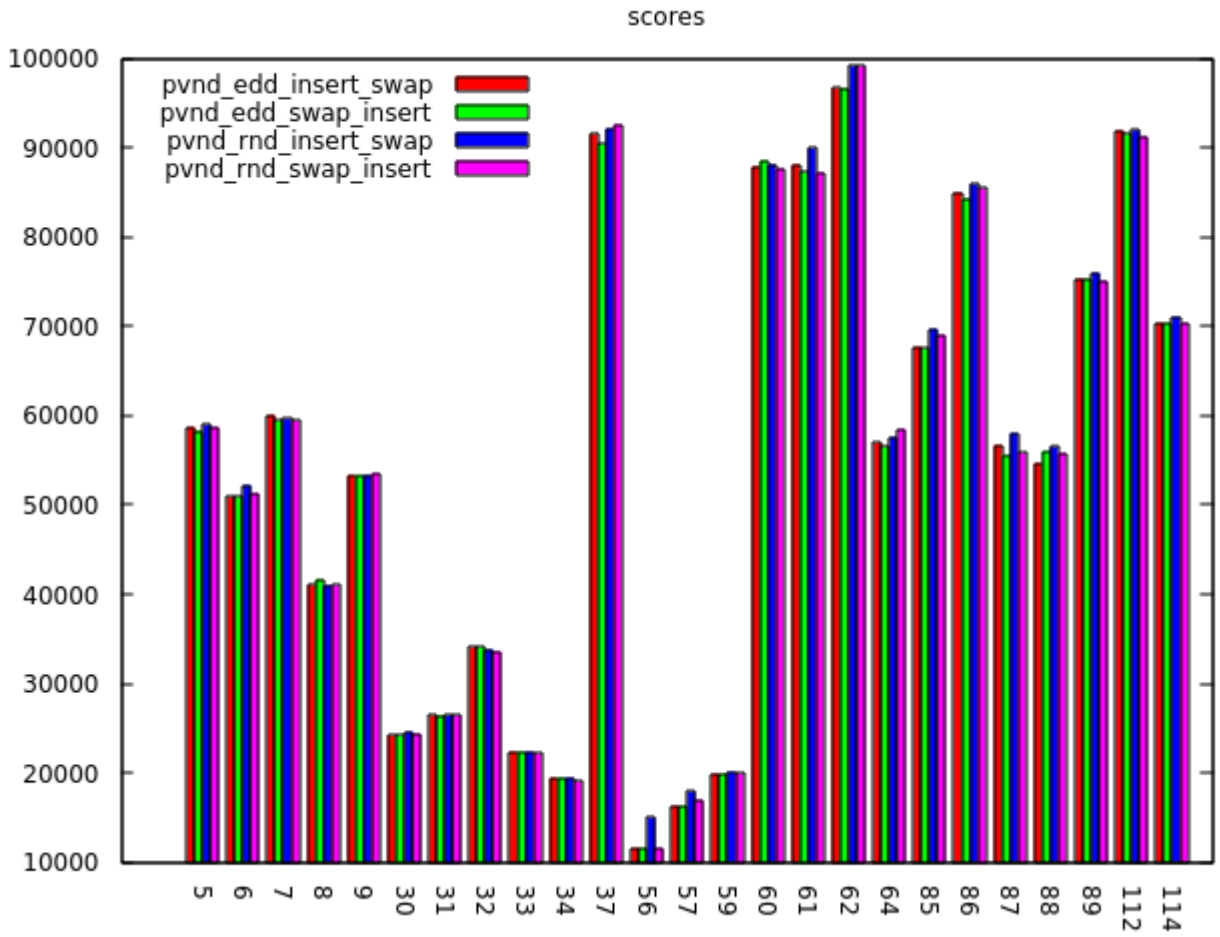
(a) Score des instances de difficultés 2 pour *edd-insert-swap* en rouge, *edd-swap-insert* en vert, *rnd-insert-swap* en bleu, *rnd-swap-insert* en rose

Les variations entres les algorithmes sont très minimes, et on peut même voir que parfois, l'heuristique de départ *rnd* donne parfois de meilleurs résultats que l'heuristique *edd* alors qu'on

pourrait pensé le contraire, car au départ elle est techniquement moins bonne. De plus, on peut voir en A.1 que les 4 versions donnent les mêmes résultats sauf sur une instance.

1.3.2 Piped Variable Neighborhood Descent (PVND)

De même que pour le VND 1.3.1, sauf que s'il on trouve une meilleure solution, on reste dans le même voisinage. Voici le graphique des résultats obtenus sur les instances de difficultés 2 :



(a) Score des instances de difficultés 2 pour *edd-insert-swap* en rouge, *edd-swap-insert* en vert, *rnd-insert-swap* en bleu, *rnd-swap-insert* en rose

1.3.3 Comparaison VND et PVND

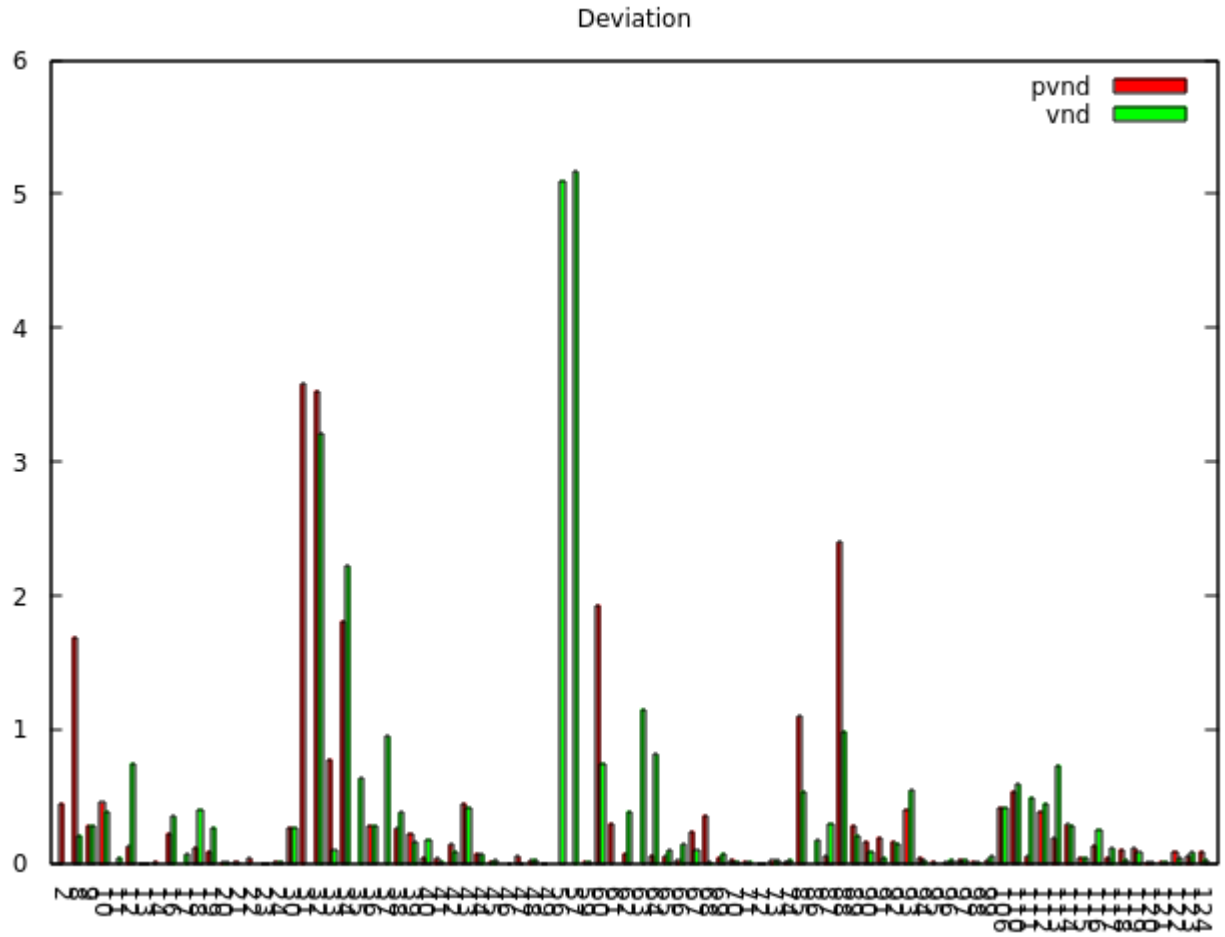
Afin de comparer les deux versions, nous allons compter combien d'optimum chaque algorithmes atteignent :

Algorithmes	Heuristique	Voisinage	Nombre d'optimum
VND	*rnd*	*insert-swap*	29
VND	*rnd*	*swap-insert*	36
VND	*edd*	*insert-swap*	45
VND	*edd*	*swap-insert*	47
PVND	*rnd*	*insert-swap*	27
PVND	*rnd*	*swap-insert*	33
PVND	*edd*	*insert-swap*	42
PVND	*edd*	*swap-insert*	48

D'après ces résultats, l'heuristique *Edd* permet d'obtenir plus d'optimum. De plus, le voisinages *swap-insert* dans cet ordre, donne également de meilleurs résultats.

Nous allons maintenant, regarder la déviation par rapport au optimum sur les instances dont on ne l'atteint pas et uniquement avec l'heuristique *Edd* et le voisinage *swap-insert*.

Voici la déviation pour le VND en rouge, et le Piped VND en vert



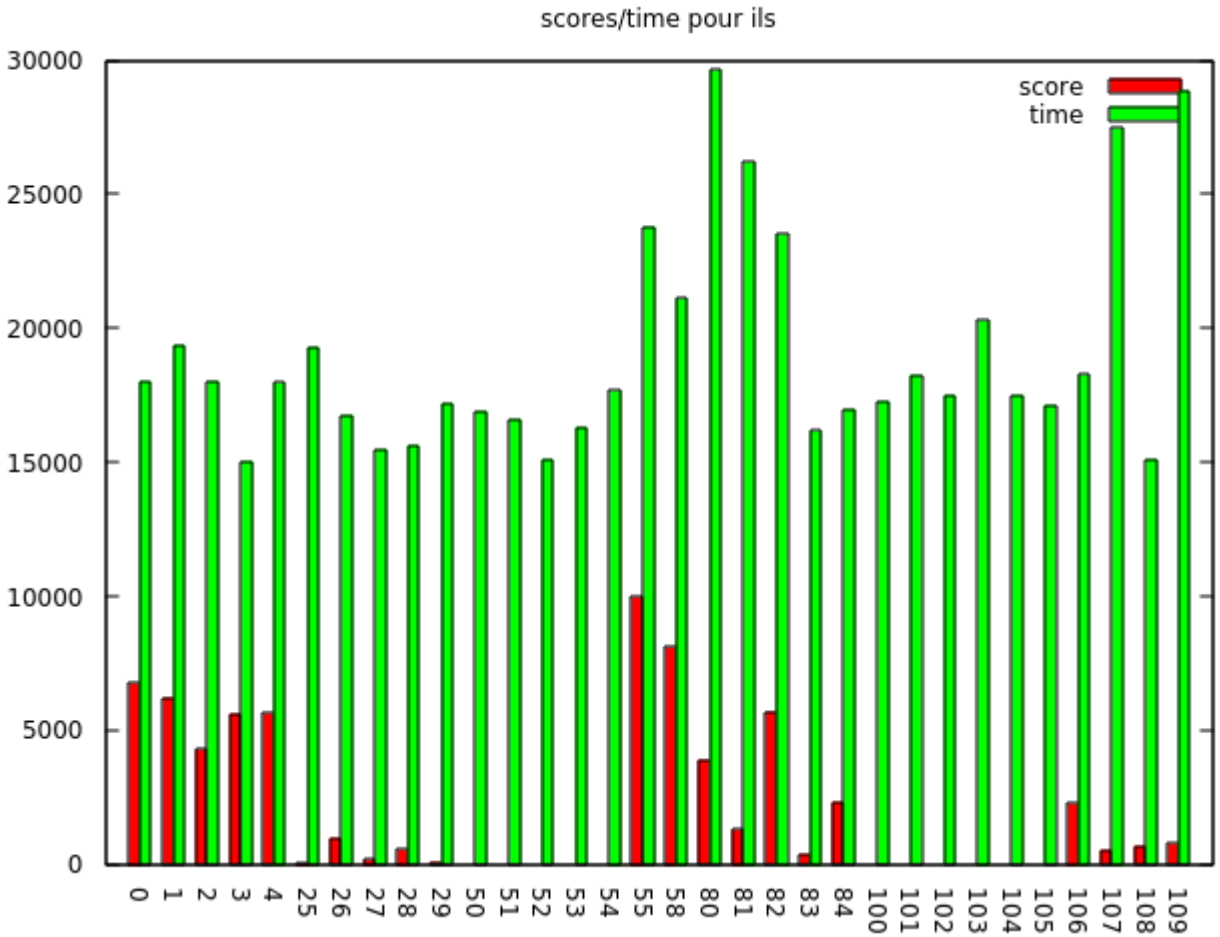
(a) Déviation pour le VND en vert, et le Piped VND en rouge

1.4 Recherche Avancée

1.4.1 Iterated Local Search (ILS)

Premiers Résultats

J'ai fait tourner une première fois mon ILS avec la configuration suivante :
— Départ *Edd*
— Recherche Locale *Best-Swap*
— Pertubation entre 3 et 12 *exchange* aléatoire
— Critère d'arrêt après au moins 15 seconde de calcul
Voici le graphique du temps et des scores obtenus sur les instances de difficulté 1 :

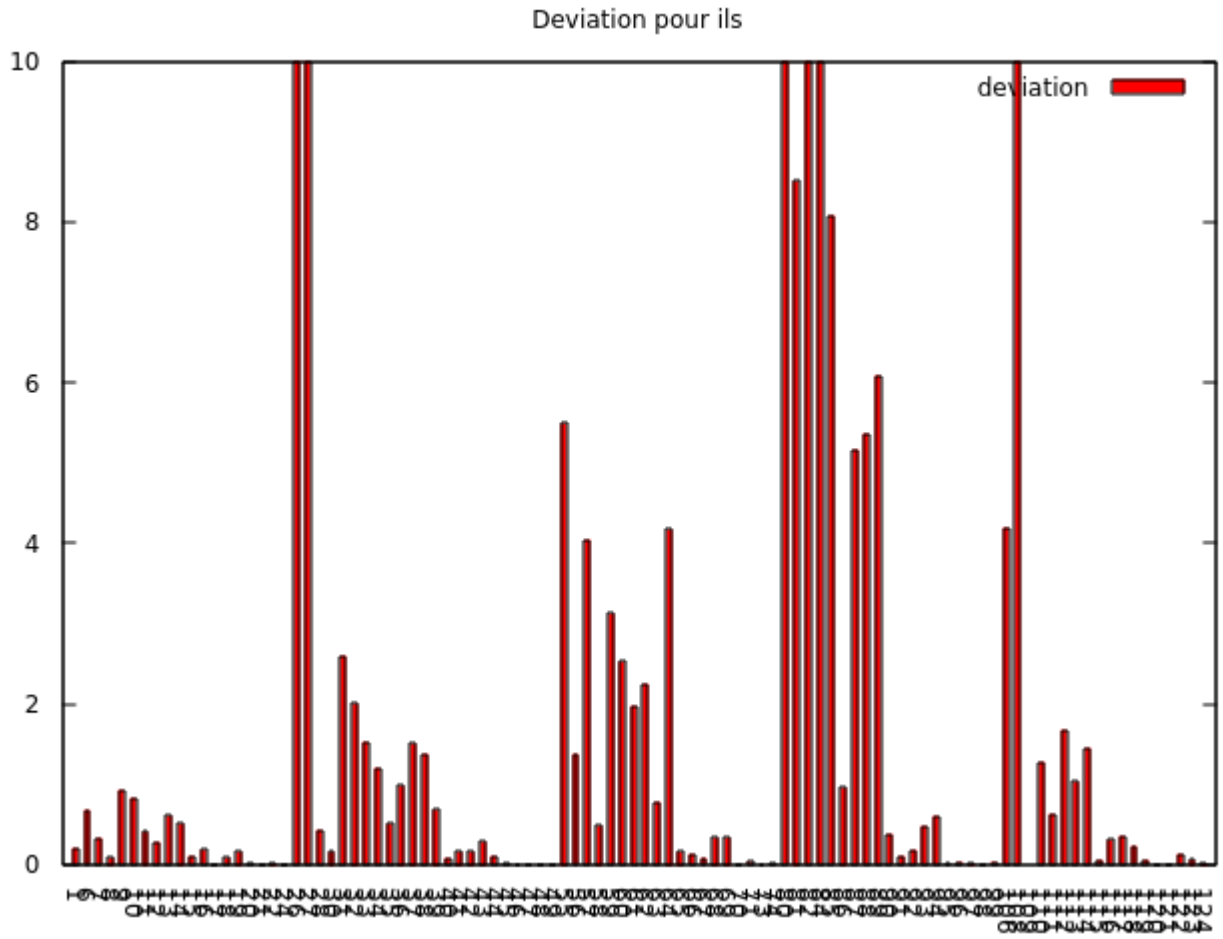


(a) Score et temps obtenus avec l'ils *Edd-Best-Swap-Exchange*-15 Secondes

J'ai pu m'apercevoir que le temps dépasse souvent 15 secondes, et que l'on n'obtient pas de bons résultats. La perturbations permet des moves vers de moins bonnes solutions pour diversifier, mais je n'ai pas garder en mémoire la meilleure solution, et donc, retourne parfois une mauvaises comparé a ce que l'ils a pu visité.

Deuxième Configuration

C'est pour quoi dans un second temps, j'ai décidé de garder la meilleure solution visitée en mémoire et j'ai également changer mon critère d'arrêt : j'accorde 30 secondes par niveau de difficulté, car les instances les plus faciles prennent moins de temps que les plus difficiles. J'ai également triché car j'arrête lorsque l'ils atteint l'optimal connus ainsi, je ne perd pas de temps de calcul sur les optimal atteint. Ainsi on obtient 27 optimums, et ce graphique des déviations :

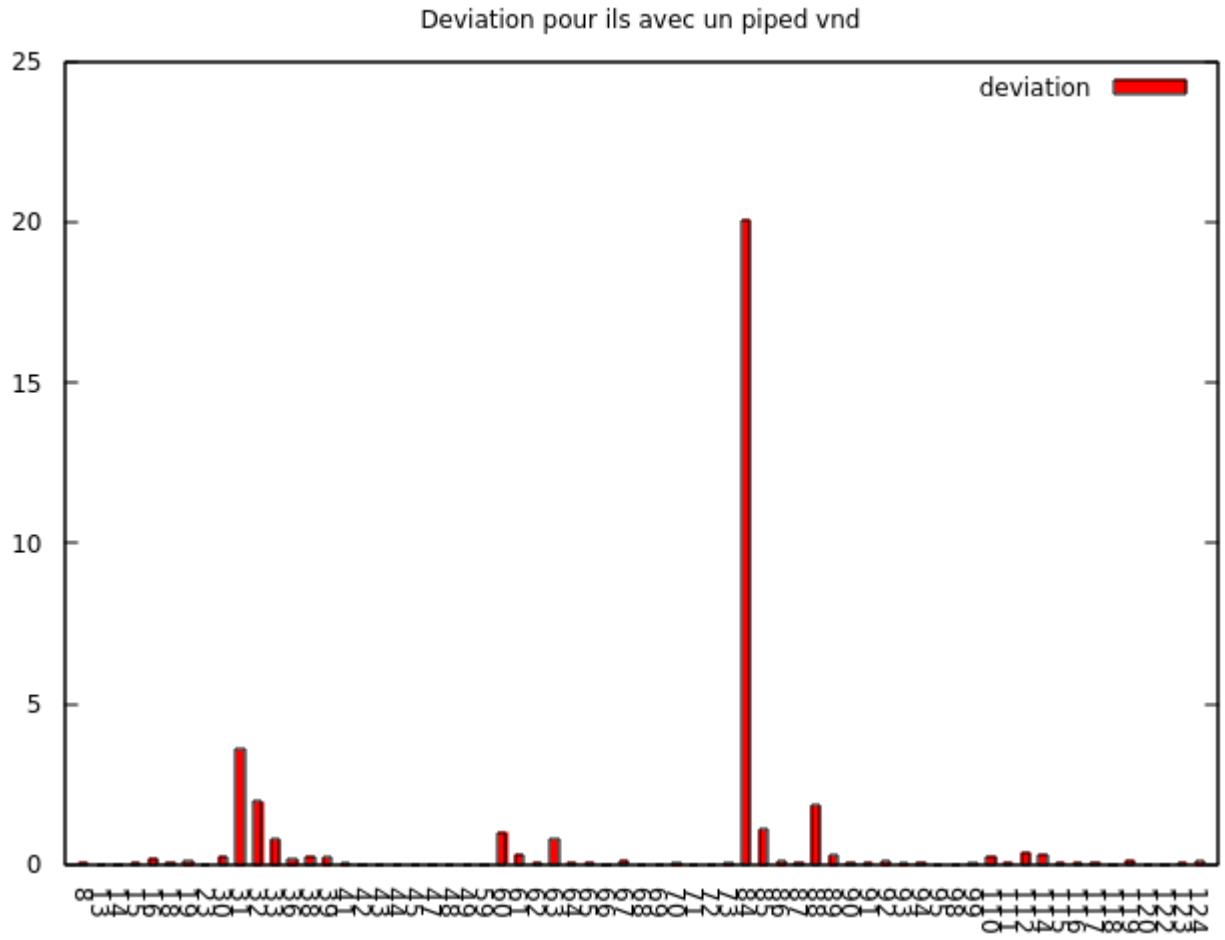


(a) Déviation par rapport a l'optimal en %

Ces résultats sont vraiment très mauvais, mais j'ai pensé que la piste était plus ou moins bonnes. J'ai donc couplé le piped vnd, dans la configuration qui a permis d'atteindre le plus d'optimum 1.3.3 et l'ils afin d'essayer d'atteindre un maximum d'optimal. La range des y est fixé à 10, mais on obtient en réalité des déviations de plus de 200% parfois.

Troisième Configuration

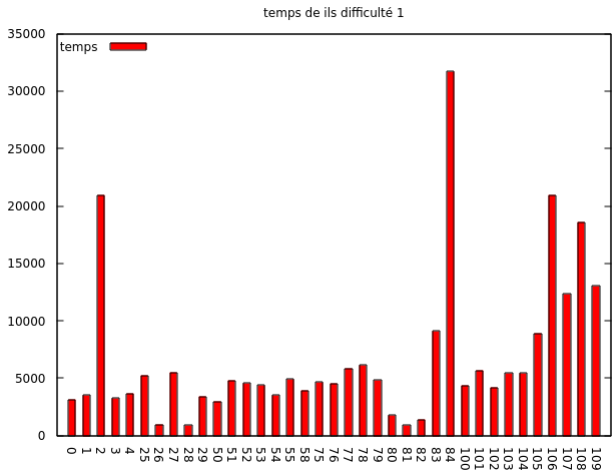
Avec le piped vnd en recherche pour l'ils, j'obtient 60 optimum connus. Voici le graphiques des déviations des 65 autres instances :



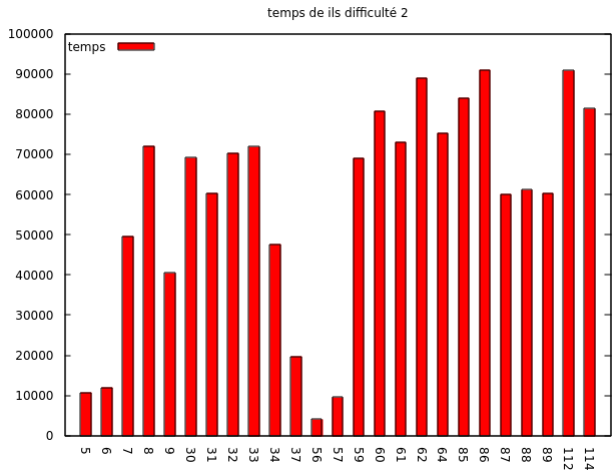
(a) Déviation par rapport a l'optimal en %

On obtient moins de 5 % de déviation sauf sur l'instance 84 où on atteint 20 %. Ces résultats sont les meilleurs obtenus pour l'instant.

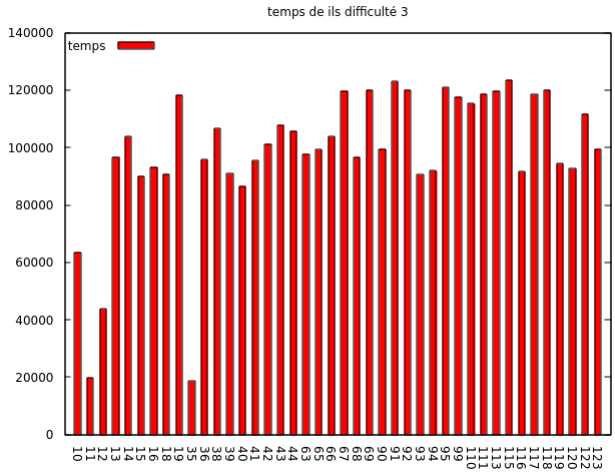
Voici les histogrammes du temps qu'à mis l'ils en fonction des 4 difficultés :



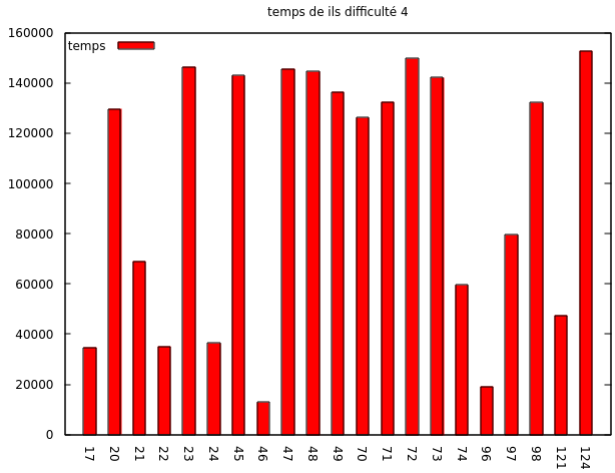
(a) Temps pour les instances de difficultés 1



(b) Temps pour les instances de difficultés 2



(a) Temps pour les instances de difficultés 3



(b) Temps pour les instances de difficultés 4

On peut voir que le fait d'arrêter lorsque l'on atteint l'optimal connus fais gagné beaucoup de temps, notamment sur les instances de difficultés 4. Peut-être que le classement n'est pas le bon, et ne reflète pas les véritables niveaux difficultés.

1.4.2 Algorithme Génétique

Voici la configuration choisie pour l'algorithme Génétique :

— Population initiale : 1 *Edd*, 1 *Mdd* et le reste en *Rnd*

— Crossover : Position Based, on forme deux enfants différents a partir de deux parents sélectionnés aléatoirement dans la population courante. On prend une portion des indices d'un parents, et on remplis le reste avec les indices de l'autre dans le même ordre.

— Mutation : On génère des nouveaux individus avec *swap* sur une portion de la population.

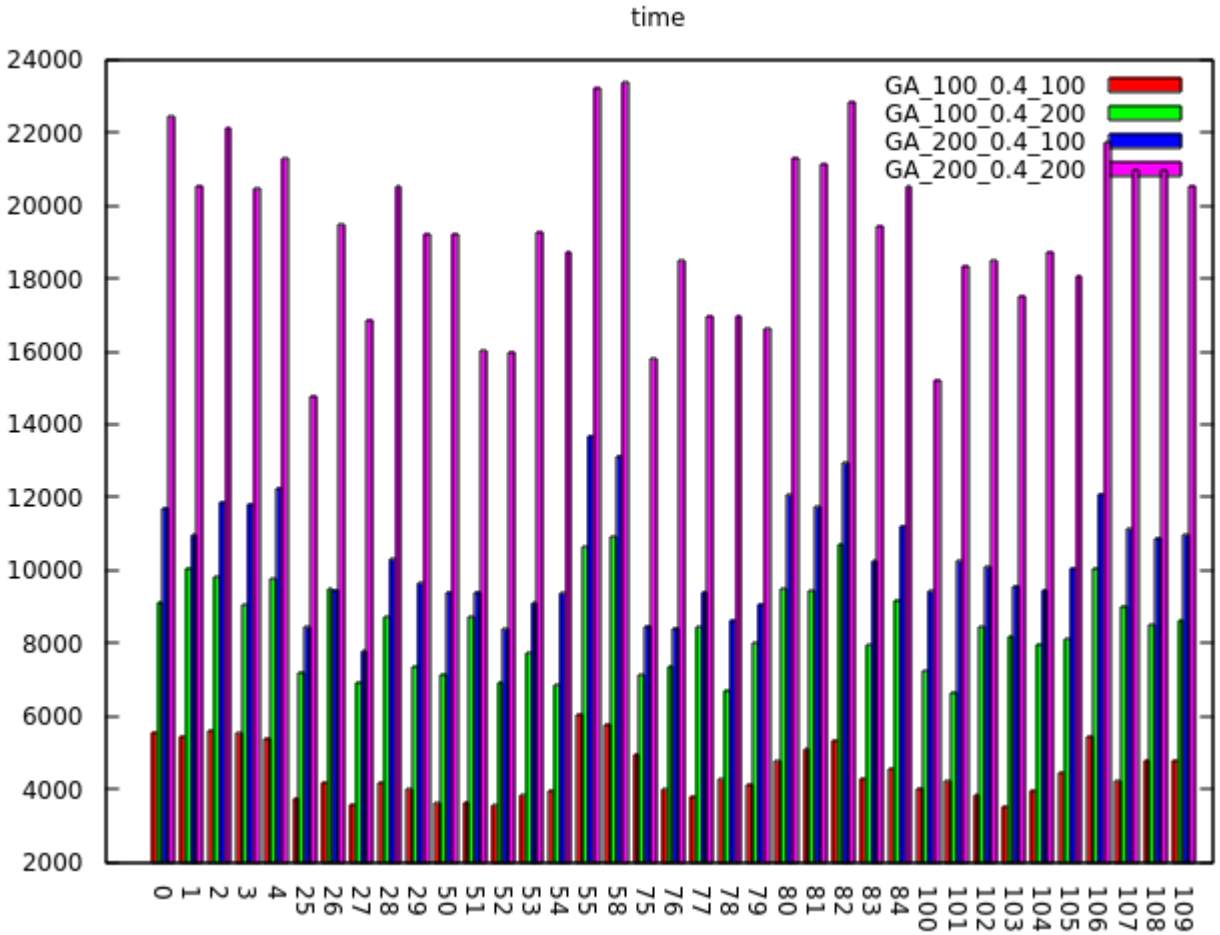
— Sélection : On prend les meilleurs individus parmi la population courante et les deux nouvelles (enfants et mutants).

— Arrêt : On fait un certain nombre de génération.

J'ai fait 4 versions de paramétrage :

Taille de population	Nombre de génération
100	100
200	100
100	200
200	200

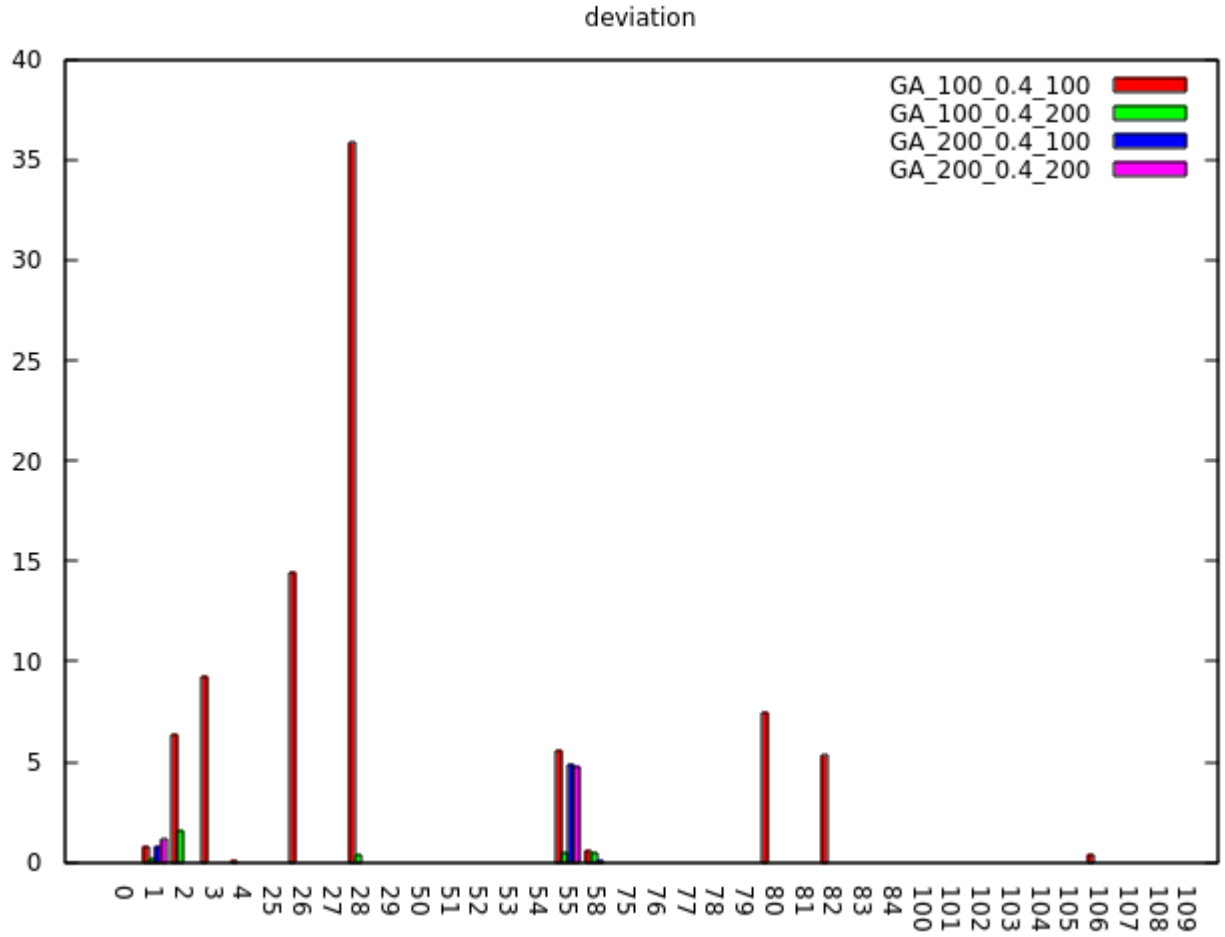
Voici, l'histogramme des temps sur les instances de difficultés 1 obtenus avec 4 algorithmes :



(a) Temps obtenus avec les 4 algorithmes sur les instances de difficultés 1

Comme on peut le deviner, plus il y a de génération, ou une grande population, plus l'algorithme prend de temps. En revanche, on peut dire qu'augmenter la population augmente plus le temps de calcul que d'augmenter le nombre de génération.

En terme de performance, on obtient de bon résultat, voici l'histogramme des déviations pour les instances de difficultés 1 :



(a) Déviation obtenus avec les 4 algorithmes sur les instances de difficultés 1

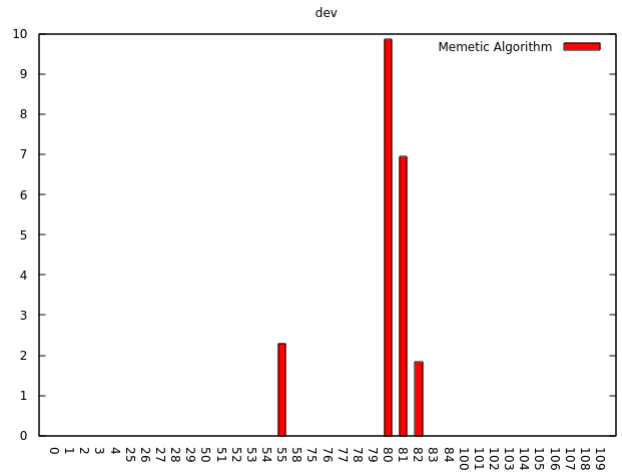
Avec 200 de population initiale et 200 générations, on obtient le best de 35 instances sur 37. [1]

1. les indices sur l'axe des x ne sont pas les indices des instances.

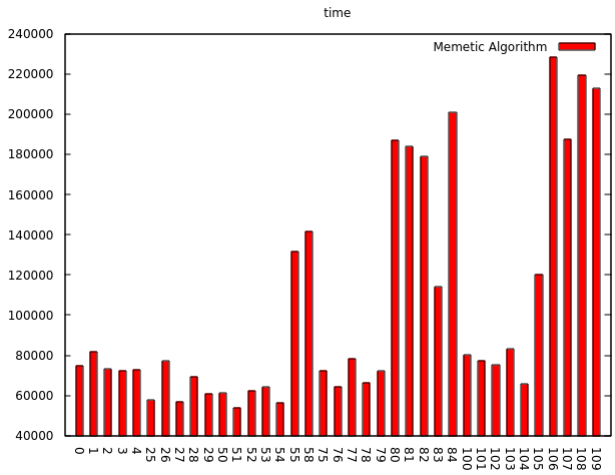
1.4.3 Algorithme Mémétique

Afin de pouvoir comparer les deux version d'algorithme génétiques, j'ai choisi d'étudier les instances de difficultés 1 car l'algorithme Mémétique consomme énormément de temps.

Voici les histogrammes du temps et de la déviation pour les instances de difficultés 1 :



(a) Déviation obtenus avec l'algorithme Mémétique sur les instances de difficultés 1



(b) temps obtenus avec l'algorithme Mémétique sur les instances de difficultés 1

On obtient également de bons résultats 33 optimums sur 37, en autant de temps que l'algorithme génétique.

2 Multi-Objectifs

2.1 Filtrages

2 Objectifs

Voici un tableau qui donne les temps d'exécution du filtre en fonction du nombre d'instances, sur une moyenne de 500 exécution pour deux Objectifs :

Filtre	Résultat				
OffLine	Nombre d'instance	Moyenne	Min	Max	Moyenne du nombre de Comparaison
	500	47	45	343	18717
	1000	128	115	884	44171
	2000	373	335	2269	103758
Online	Nombre d'instance	Moyenne	Min	Max	Moyenne du nombre de Comparaison
	500	164	69	294	41381
	1000	364	135	1202	98434
	2000	745	300	2518	225416

3 Objectifs

Voici un tableau qui donne les temps d'exécution du filtre en fonction du nombre d'instances, sur une moyenne de 500 exécution pour trois Objectifs :

Filtre	Résultat				
OffLine	Nombre d'instance	Moyenne	Min	Max	Moyenne du nombre de Comparaison
	500	60	57	329	51614
	1000	154	142	458	131990
	2000	461	406	826	327939
Online	Nombre d'instance	Moyenne	Min	Max	Moyenne du nombre de Comparaison
	500	460	167	960	225038
	1000	1063	254	1952	584937
	2000	2558	958	4636	1568141

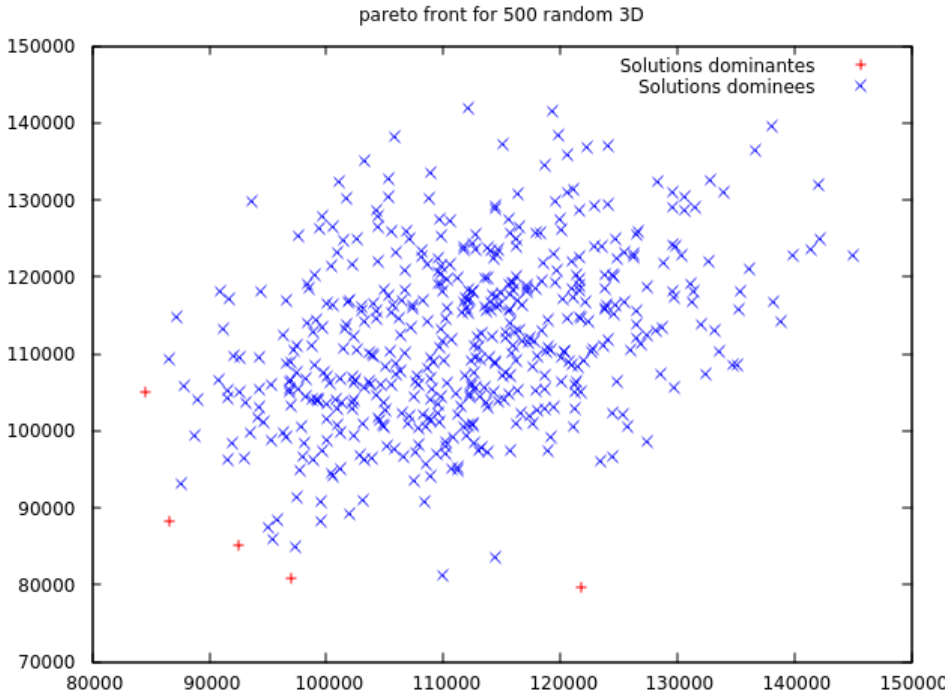
Remarques

On peut dire que mon filtre Offline est plus performant devant mon filtre Online.

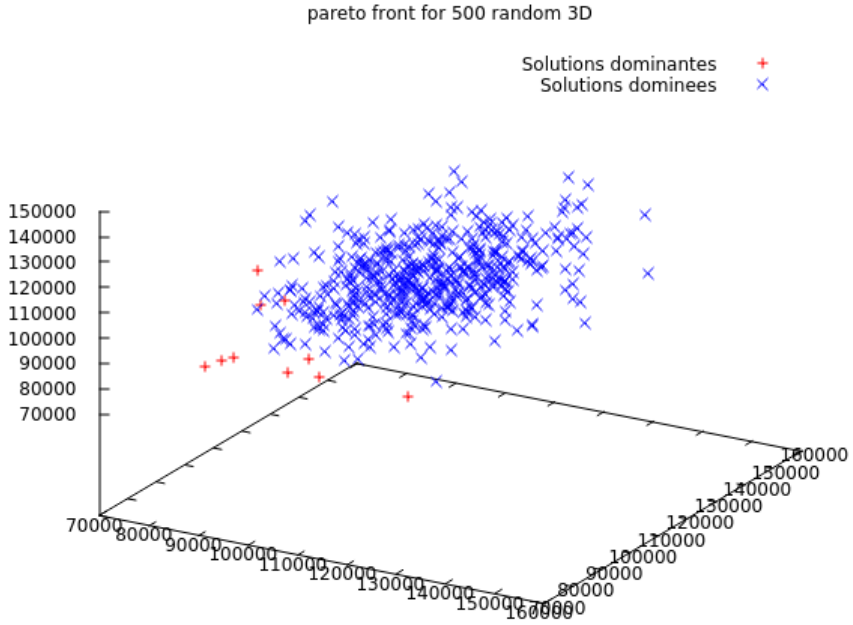
Je ne sais pas pourquoi, mais pour le filtre OffLine, la première exécution est beaucoup plus longue que les autres (le maximal), et les autres sont toutes proches du minimal. J'ai pourtant bien remis à zéro toutes les listes, verifier que les générations se font de 0, mais rien ne fait. Peut-être une optimisation de la JVM. Hors mis cette remarque, on peut dire également que l'écart-type du filtre Online est plus grand que le filtre Offline

Graphiques

Voici un graphique pour 500 instances Random pour deux et trois Objectifs. En rouge, le front Pareto, en bleu les solutions Dominées :



(a) Front pareto en rouge, et solutions dominées en bleus pour deux objectifs



(a) Front pareto en rouge, et solutions dominées en bleus pour deux objectifs

2.2 Algorithmes pour mTSP

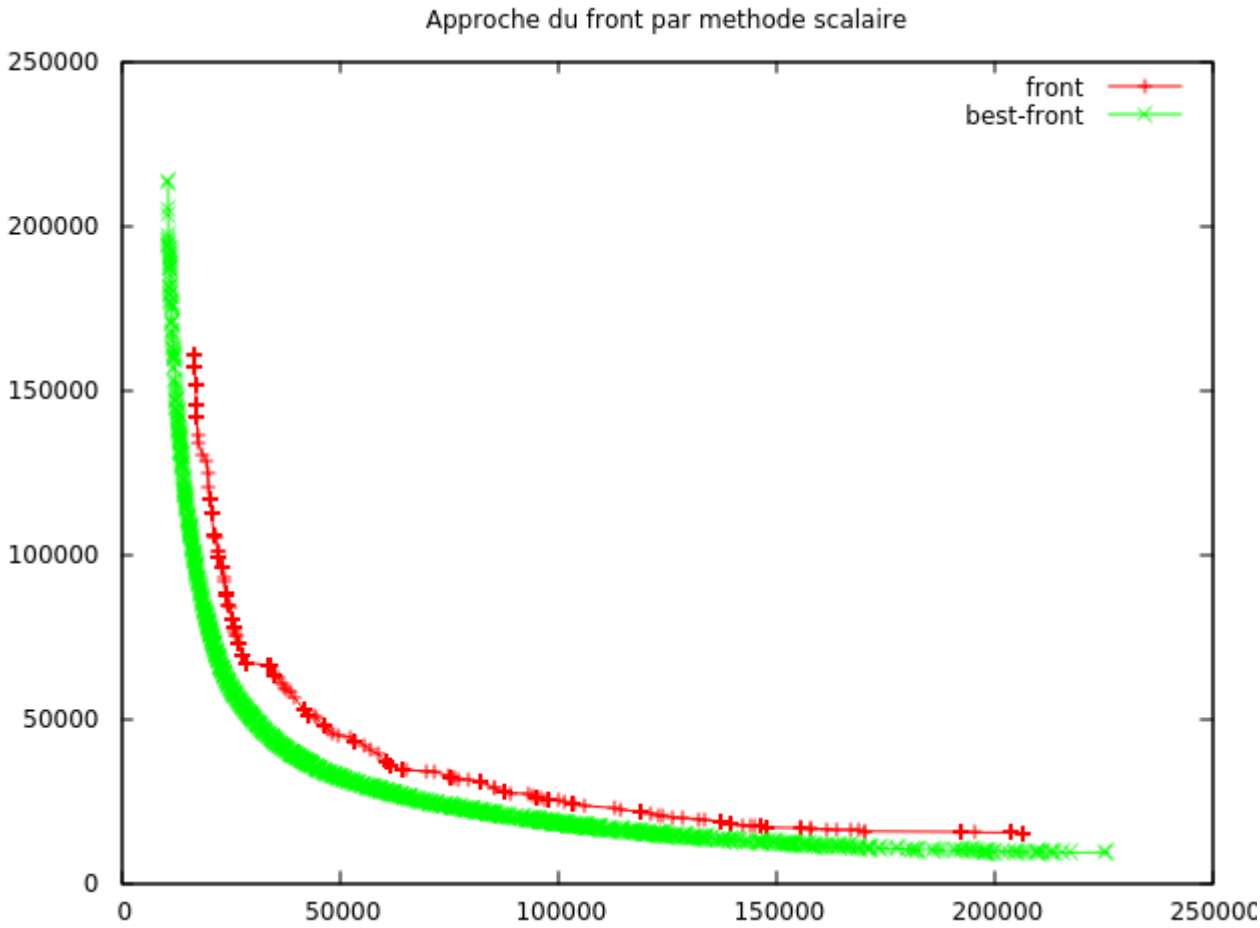
Dans cette section, nous allons étudier deux techniques pour approché le front Pareto : l'approche Scalaire et l'approche Pareto.

2.2.1 Approche Scalaire

Voici la configuration de l'approche Scalaire utilisée :

— Heuristique de départ : *Rnd*
— Sélection : *Best*
— Voisinage : *2-opt*
— Vecteur de Poids : <0.0,4000.0> pas de 1.0
— Mis a jour des poids : Lorsque l'on a atteint un optimal local, on change les poids($+$ *pas* en x, et $-$ *pas* en y.
— A chaque changement de poids, on utilise le meilleur voisin de la solution retenue.
— Arrêt : Lorsque que l'on a parcouru tout le vecteur poids.

Voici le front Pareto obtenu grâce à l'approche Scalaire pour deux objectifs obtenus en 2704575 ms :



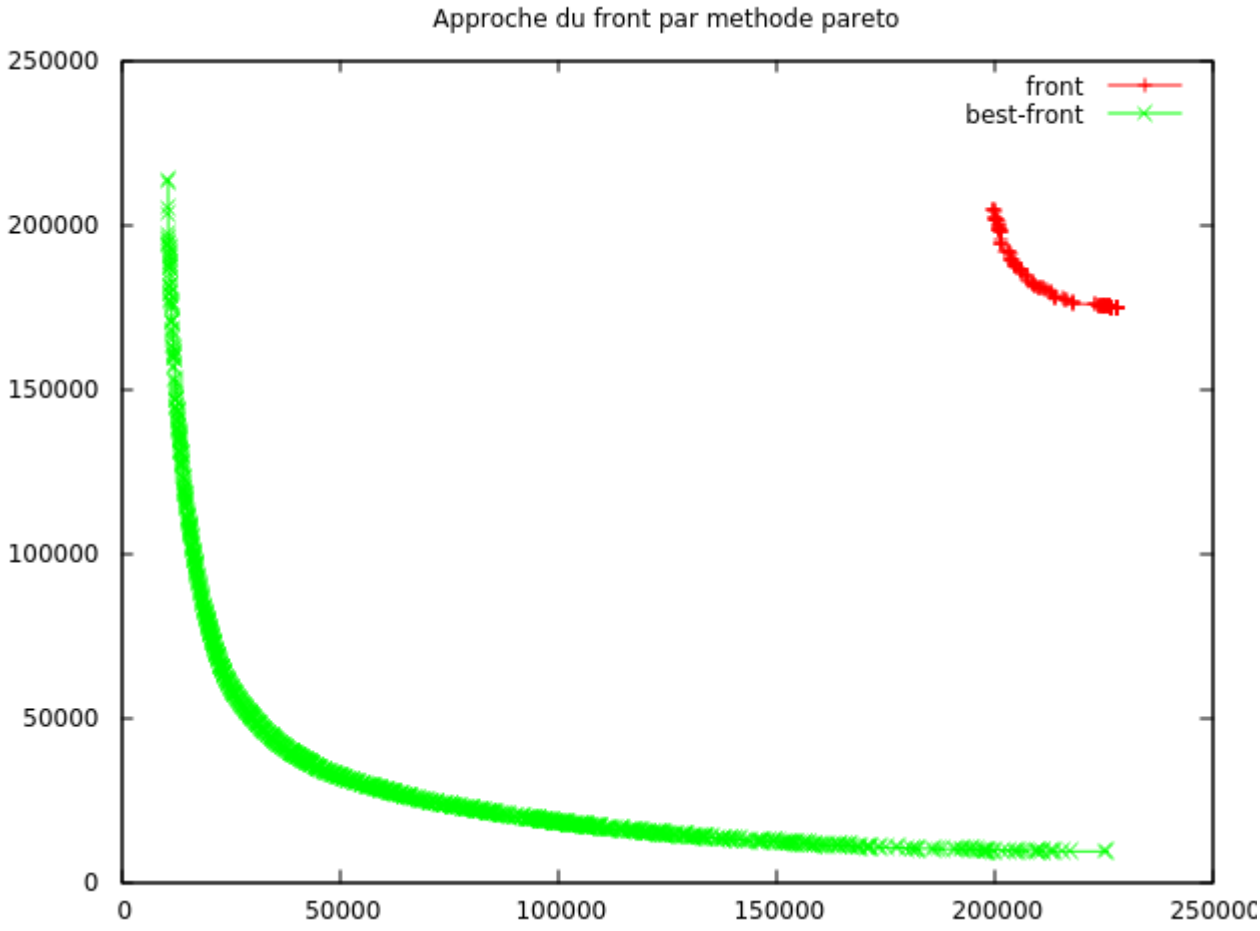
(a) Front pareto obtenus par approche Scalaire pour deux objectifs, en rouge le front de la méthode scalaire, en vert le front du *best-known*

2.2.2 Approche Pareto

Voici la configuration de l'approche Pareto utilisée :

— Heuristique de départ : une population de solutions *Rnd*
— Voisinage : *2-opt*
— Sélection : On prend une par une les solutions de départ, on génère le voisinage, et on garde que les solutions non-dominées qu'on mets dans l'archive.
— Arrêt : Lorsque l'on a visité chacune des solutions de départ.

Voici le front Pareto obtenu grâce à l'approche Pareto avec la même heuristique que celle pour l'approche Scalaire, j'ai laisser tourner autant de temps que l'approche scalaire :



(a) Front pareto obtenus par approche Pareto pour deux objectifs, en rouge le front de la méthode Pareto, en vert le front du *best-known*

Comparaison des deux algorithmes

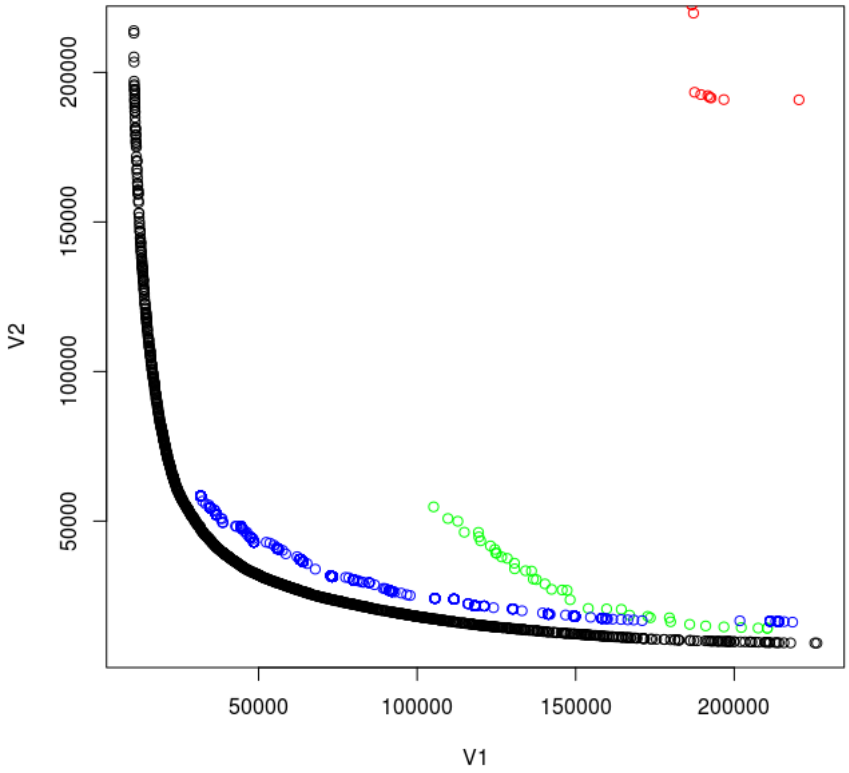
Le seul aléatoire inclus dans les algorithmes sont les indices des générations de voisins, or comme on visite tout le voisinage, cet aléatoire n'a aucune influence sur eux, donc ils sont déterministes. (et les solutions initiales mais on utilise la même).

Il va être difficile de comparait les deux algorithmes, car les conditions initiales ne sont pas les mêmes. J'ai tenté d'utilisé la même solution de départ de l'approche Scalaire en guise de population pour l'approche Pareto, mais les résultats ne semblent pas très bon.

On peut déjà dire d'après les sections 2.2.1 et 2.2.2, que mon approche Scalaire est meilleure en performance et en temps que mon approche Pareto. En revanche, j'ai pu avec la méthode Pareto, approcher le front pour 3 objectifs. (voir l'annexe B.1.1)

2.2.3 Hybridation des algorithmes

J'ai croisé les deux approches, afin de tenter d'avoir de meilleurs résultats. Voici la configuration utilisée : On effectue une approche scalaire sur la tête de la liste, puis on effectue une génération de tous les voisins. On filtre la concaténation des deux résultats, et on ajoute les solutions dominantes à l'archives et au solutions génératrices. Pour pouvoir brider les deux approches, on réduit le vecteur de poids à <0.0,10.0> par pas de 0.5. Un nouveau critère d'arrêt est requis, c'est pour quoi j'ai accordé 1 000 000 ms à l'algorithme. Voici le résultats, comparer aux approches simples et au *best-known* :



(a) En rouge, l'approche Pareto, en vers l'approche Hybride, en bleu l'approche Pareto et en noir le *best-known*

Veuillez trouver un graphe qui confirme les tendances des algorithmes sur différentes solutions de départ en Annexe B.1.2

L'approche Hybride manque de diversité, je pense que ce problème vient du petit vecteur de poids utilisé lors de l'approche Scalaire dans l'approche Hybride.

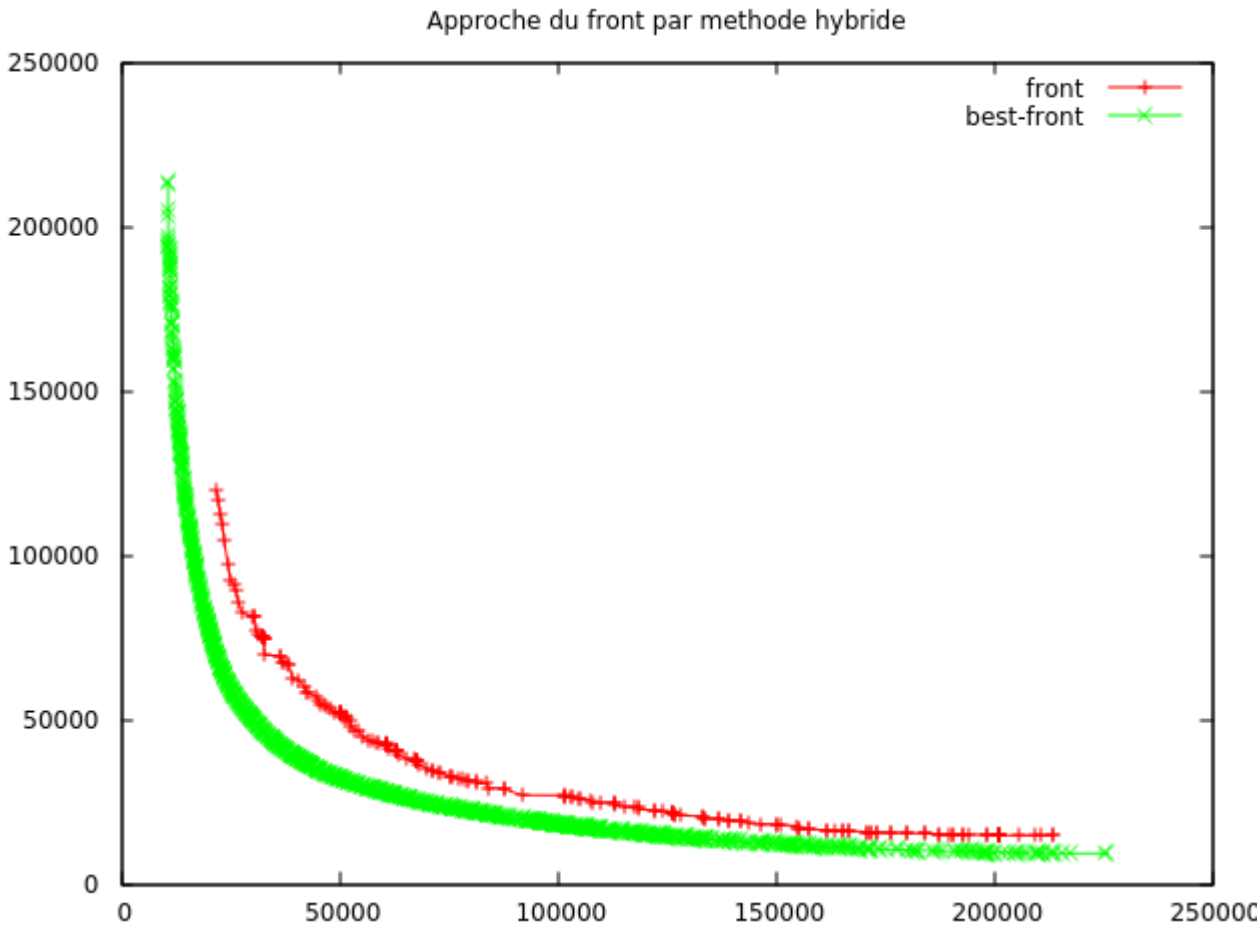
2.2.4 Hypervolume

On va ici, comparer les hypervolumes des solutions trouvées. On utilise les valeurs trouvées à la section 2.2.3 pour pouvoir comparé (mêmes conditions initiales, même condition d'arrêt).

Algo	Hypervolume	Déviation
best-known	40872050437	0.0
Pareto	1031273234	0.974
Scalaire	6096441521	0.851
Hybride	2956852545	0.927
$Scalaire_1$	24502016837	0.400
$Pareto_1$	647815156	0.984
Hybride 2	17064304135	0.582

$Scalaire_1$ fait référence au premier calculer à la section 2.2.1, de même que pour $Pareto_1$ pour la section 2.2.2

J'ai effectué un nouveau run de l'hybride, avec plus de temps (autant que les pour les autres 2.2.1 et 2.2.2) avec un vecteur de poids et un pas plus grand. Voici le graphique obtenu :



(a) Front pareto obtenus par approche Pareto pour deux objectifs, en rouge le front de la méthode Pareto, en vert le front du *best-known*

3 Conclusion

J'ai pu à travers ce projet, me plonger un peu plus l'optimisation combinatoire que j'avais déjà découvert en M1. J'ai également appris la difficulté de crée des expérimentations pertinentes et performantes, ainsi que l'exploitation de leurs résultats. J'ai perdu du temps sur des problèmes de conceptions et de développent, mais cela m'a permit d'apprendre à travailler sur un gros projet.

A Mono-Objectif

A.1 VND

On peut voir ici que les 4 algorithmes donnent les mêmes résultats sauf pour une seule instance :

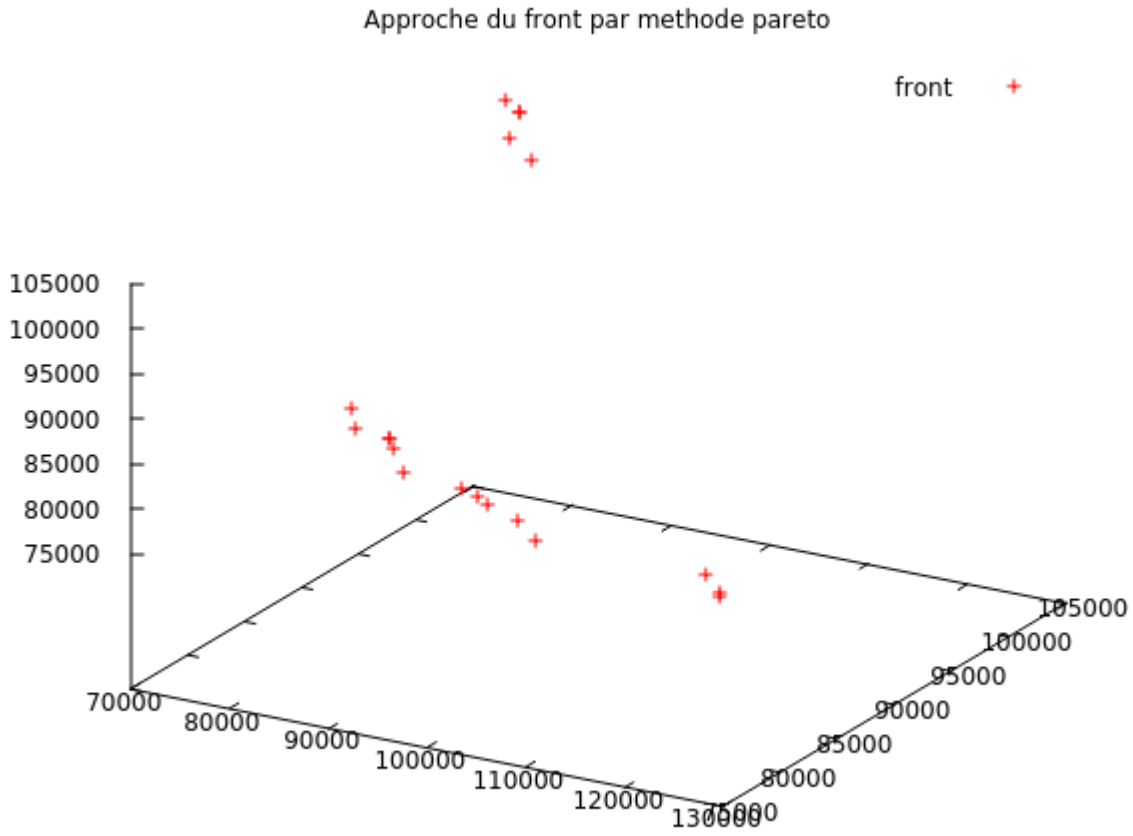
(a) Score des instances de difficultés 4 pour *edd-insert-swap* en rouge, *edd-swap-insert* en vert, *rnd-insert-swap* en bleu, *rnd-swap-insert* en rose

B Multi-Objectif

B.1 Algorithme pour mTSP

B.1.1 Approche Pareto

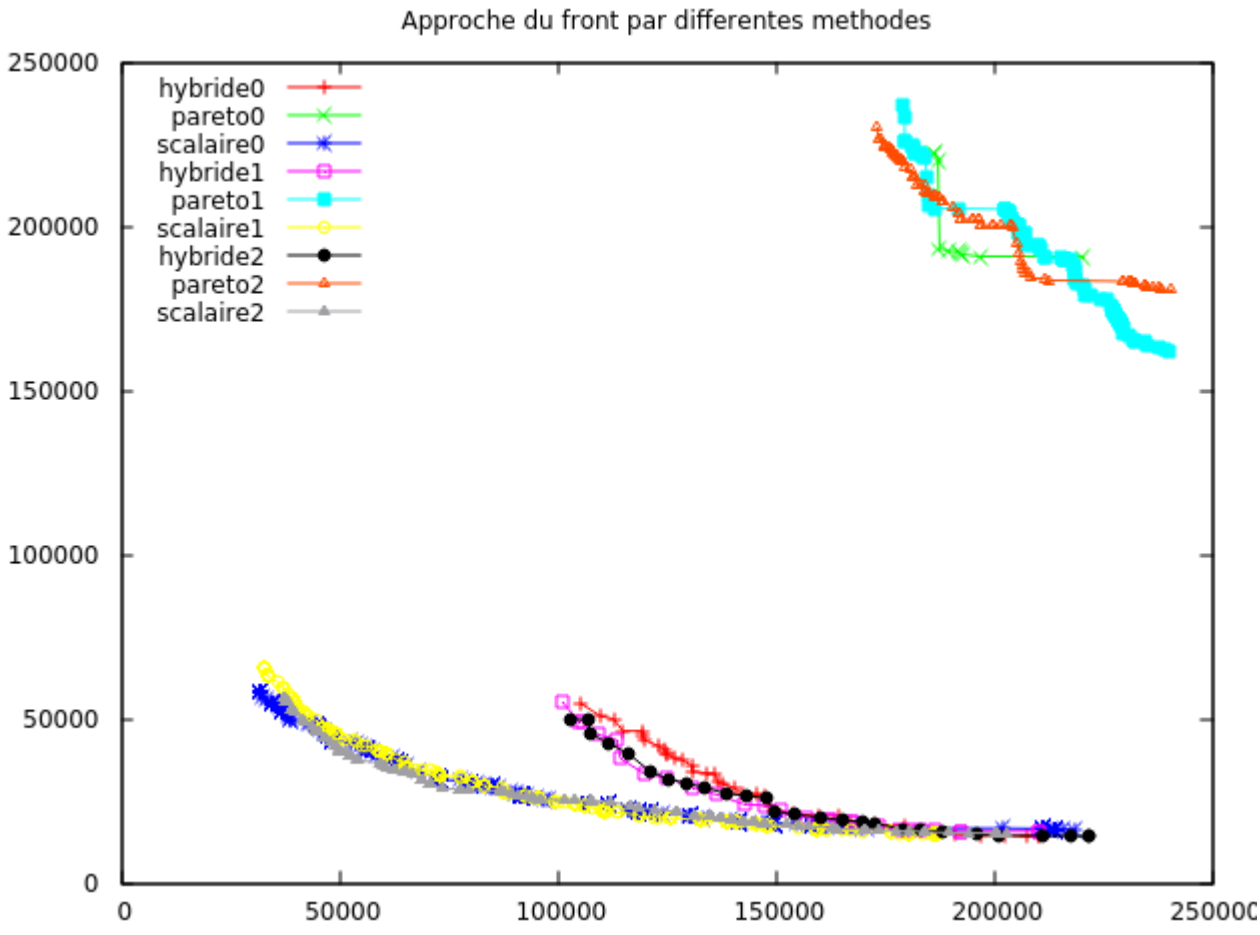
Voici le front Pareto obtenus pour 3 Objectifs dans les mêmes conditions décrites à la section 2.2.2 obtenus en 94565 ms :



(a) Front Pareto 3 Objectifs

B.1.2 Hybridation des algorithmes

Voici les fronts obtenus par les 3 méthodes :


Approche du front par differentes methodes
hybride0
pareto0
scalaire0
hybride1
pareto1
scalaire1
hybride2
pareto2
scalaire2
250000
200000
150000
100000
50000
0
0
50000
100000
150000
200000
250000

(a)