POLYTECHNIQUE MONTRÉAL

UNIVERSITÉ D'INGÉNIERIE

École Polytechnique Montréal

INF6102 - Métaheur. applic. au génie informatique

Hiver 2025

Devoir 3
Planification de la production alimentaire

2310478 - Hugo Mouton
2311468 - Jean Barreto

21 Avril 2025

Table des matières

1 Architecture de l'algorithme **2**

2 Solution initiale **2**
2.1 Naïve 2
2.2 Meilleure parmi K aléatoires 2
2.3 Construction itérée 3
2.4 Comparatif 3

3 Recherche Tabou **3**
3.1 Espace de recherche 3
3.1.1 Voisinages considérés 3
3.1.2 Connectivité de l'espace de recherche 4
3.1.3 Contraintes du problème 4
3.2 Fonctions de recherche locale 4
3.2.1 Fonction de sélection 4
3.2.2 Fonction d'évaluation 4
3.3 Mécanismes tabou 4
3.4 Recherche des meilleurs hyperparamètres 4
3.5 Conditions d'arrêt 5

4 Mécanisme de Restart **5**

5 Résultats & Discussion **6**
5.1 Résultats comparatifs 6
5.2 Évolution du score obtenu selon le nombre d'itération 6
5.3 Discussion 8

1 Architecture de l'algorithme

Le fonctionnement global de l'algorithme est représenté dans le diagramme ci-dessous. On construit d'abord une solution initiale avant de lui appliquer une succession de recherche tabou puis de restarts jusqu'à ce que le temps soit écoulé, ou que la valeur de la borne inférieure soit obtenue, on retourne alors la meilleure solution trouvée. Chaque bloc sera plus amplement développé dans les sections suivantes.

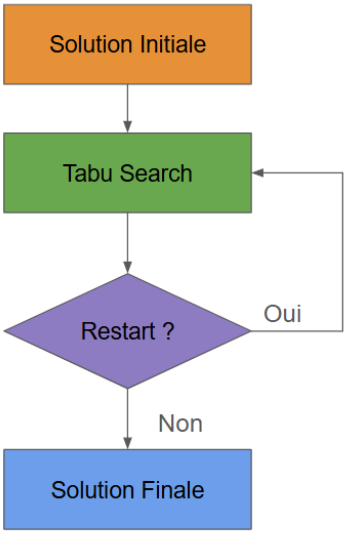


FIGURE 1 – Architecture globale

2 Solution initiale

Dans cette section nous présentons trois façons de construire notre solution initiale, puis nous les comparons afin de choisir la plus pertinente.

2.1 Naïve

Cette construction se base sur l'implémentation naïve fournie dans le devoir.

Complexité : Cette construction est en $\mathcal{O}(J \times C)$ (deux boucles *for* imbriquées).

2.2 Meilleure parmi K aléatoires

La construction décrite ci-dessous génère K solutions aléatoires valides et retourne la meilleure :

1. On initialise la solution avec des -1, c'est-à-dire sans production.
2. On parcourt les jours j dans l'ordre.
3. Pour chaque jour j on parcourt les configurations c.
4. Pour chaque couple (j, c) on regarde si un produit de configuration c est requis le jour j.
5. Si oui, on place la configuration c aléatoirement dans un jours $j', j' < j$ encore vide.

Complexité : Cette construction à une complexité comprise entre $\mathcal{O}(J \times C \times K)$ et $\mathcal{O}(J^2 \times C \times K)$. Cet intervalle est dû à la dernière boucle permettant de mettre à jour une structure de données utile, mais possédant un nombre d'itération

stochastique qu'il n'est pas nécessaire de quantifier au vu des faibles valeurs prises par J sur les différentes instances. La complexité reste donc raisonnable dans les deux cas.

2.3 Construction itérée

Cette construction se déroule comme suit :

1. On initialise la solution avec des -1, c'est-à-dire sans production.
2. On parcourt les jours j dans l'ordre, et pour chaque j :
   (a) On parcourt les configurations c.
   (b) Pour chaque couple (j, c) on regarde si un produit de configuration c est requis le jour j.
   (c) Si oui, on place la configuration c aléatoirement dans un jours $j', j' < j$ encore vide.
   (d) Après être passé sur les C configurations on réalise un *Hill Climbing* en effectuant des 2-Swaps de jours de production, on s'arrête lorsque le minima est rencontré.

Complexité : Cette construction à la même complexité que la construction précédente au nombre moyen de *Hill Climbing* effectué près.

2.4 Comparatif

Le tableau ci-dessous présente les résultats obtenus pour ces trois modèles de génération de solution initiale sur une seed fixée :

Instance	Naive	Meilleure parmi K	Construction itérée
A	3145 | 0.00	2154 | 00.00	1954 | 0.00
B	19268 | 0.00	17165 | 28.24	12670 | 0.40
C	56042 | 0.00	49957 | 39.41	29138 | 3.55
D	38870 | 0.00	36454 | 40.23	21592 | 2.59
E	63720 | 0.00	51338 | 57.76	18239 | 6.19
F	55074 | 0.00	46888 | 58.86	20598 | 6.03

TABLE 1 – Comparaison des résultats des solutions initiales avec leur temps d'exécution (en sec).

Au vu des résultats ci-dessus il semblerait qu'une solution initiale basée sur le modèle de la **Construction Itérée** soit à la fois plus rapide et de meilleure qualité que les modèles **Naïf** et **Meilleure parmi K aléatoires**. Ainsi nous retiendrons ce modèle pour notre algorithme final.

3 Recherche Tabou

Notre phase de recherche locale est constituée d'une recherche tabou.

3.1 Espace de recherche

3.1.1 Voisinages considérés

Nous considérons ici un seul type de mouvement :
Échange de productions entre 2 jours - `neighbors_2_swap` : Pour chaque $(i, j) \in \{J^2 \mid i < j\}$ on échange la configuration en i et en j si ce mouvement conserve la validité de la solution. On peut ainsi échanger 2 productions de produits ou déplacer la production d'un produit à un jour sans production. Dans le pire des cas il y a un total de $\binom{J}{2}$ voisins. En réalité certains ne sont pas considérés car ils mènent à une solution non valide. De plus l'hyperparamètre $RATIO\_NEIGH$ permet de ne prendre qu'un pourcentage de ces voisins.

3.1.2 Connectivité de l'espace de recherche

Ce voisinage est évidemment connecté car si l'on connaît une solution optimale, on peut utiliser la fonction `neighbors_2_swap` plusieurs fois de manière appropriée afin de se rendre à la solution optimale depuis n'importe quelle solution valide.

3.1.3 Contraintes du problème

Pour notre recherche locale, les deux contraintes du problème sont dures. En effet notre solution initiale génère une solution pour laquelle tous les produits demandés sont produits et pour laquelle il n'y a aucun retard de livraison. Par la suite, nos mouvements locaux conservent ces contraintes à chaque déplacement.

3.2 Fonctions de recherche locale

3.2.1 Fonction de sélection

Le voisin sélectionné correspond au meilleur voisin améliorant, non banni par la liste tabou, dans le voisinage de la solution courante. Le nombre de voisins regardés est $RATIO\_NEIGH \times \binom{J}{2}$.

3.2.2 Fonction d'évaluation

Nous évaluons les différentes solutions en fonction de leur score. Cependant nous utilisons `change_of_storage_cost(i, j)` et `change_of_trans_cost(i, j)` afin de calculer les variations de coût engendré par un mouvement $i \leftrightarrow j$ ce qui nous évite de passer par la fonction donné dans le fichier `utils.py` qui possède une complexité en $\mathcal{O}(J \times C)$.

En effet `change_of_storage_cost(i, j)` est en $\mathcal{O}(1)$ et `change_of_trans_cost(i, j)` est en $\mathcal{O}(G)$ où G est le nombre moyen de saut à droite et à gauche de i et j qu'il faut faire pour tomber sur un jour sans production, cette fonction est donc en $\mathcal{O}(J)$. Cependant cette complexité correspond à la complexité dans le pire cas, dans le cas général, le nombre de saut moyen G est rarement égal à J et la compexité s'assimile plutôt à du $\mathcal{O}(1)$

3.3 Mécanismes tabou

Nous avons mis en place un mécanisme tabou reposant sur les trois aspects suivants :

1. **Abstraction d'un mouvement :** Pour tout mouvement $i \leftrightarrow j$ nous associons l'abstraction (i, j, c_i, c_j) où c_i et c_j sont les configurations présentes en i et j avant le mouvement.
2. **Structure tabou :** Nous mettons en place un dictionnaire tabou dont les clés sont toutes les abstractions de mouvements possibles, et où les valeurs sont l'itération jusqu'à laquelle le mouvement associé à l'abstraction qu'est la clé est banni. On initialise le dictionnaire avec des valeurs à 0. Cela nous donne un dictionnaire possédant $\binom{J}{2} \times C^2$ clés. Cela reste conséquent mais ne prend pas plus de 1 à 2 secondes à être initialisé pour les plus grosses instances.
3. **Mise à jour de la structure :** Lorsque l'on effectue un mouvement on incrémente de **TABU_LENGTH** la valeur de la clé de son abstraction. **TABU_LENGTH** est donc un hyperparamètre dont nous discuterons plus loin.

3.4 Recherche des meilleurs hyperparamètres

Dans cette section nous allons essayer de déterminer la valeurs optimale des hyperparamètres **TABU_LENGTH** et **RATIO_NEIGH** en effectuant les compromis suivants :

1. Il nous semble que, pour avoir des résultats pertinents et comparables, il soit nécessaire de faire tourner une instance sur son temps d'éxecution complet. Cependant avec 5 minutes par couple de valeurs d'hyperparamètres cela nous

aurait pris trop de temps de tester pour toutes les instances. C'est pour cela que l'on s'intéresse ici qu'à l'instance D qui est une instance moyenne en terme de taille par rapport aux autres instances. En fixant la seed à 42 pour tous les tests, nous allons donc déterminer le meilleur couple d'hyperparamètres pour l'instance D et conserver ce couple pour les autres instances.

2. Nous supposons également que la fonction associant un couple d'hyperparamètre au coût optimal est convexe ce qui nous permet d'en figer un pour optimiser l'autre et vice-versa.

Cette méthode n'est pas optimale mais cela nous permet d'avoir des résultats acceptables en un temps raisonnable.

TABU_LENGTH	Score obtenu
0	20223
10	20199
30	20016
50	19933
75	20084
100	20105

TABLE 2 – Score obtenu sur l'instance D en fonction de **TABU_LENGTH** et **RATIO_NEIGH** fixé à 0.1.

RATIO_NEIGH	Meilleur Cout
0.01	21773
0.05	19826
0.1	19553
0.2	20107
0.3	20322
0.5	20351

TABLE 3 – Score obtenu sur l'instance D en fonction de **RATIO_NEIGH** et **TABU_LENGTH** fixé à 20000.

Les résultats présents dans les tableaux ci-dessus démontrent que les meilleurs scores sont obtenus lorsque **TABU_LENGTH vaut 50** et que **RATIO_NEIGH vaut 0.1**. En considérations des compromis énnoncés plus haut nous choisiront donc ces valeurs d'hyperparamètres pour toutes les instances.

3.5 Conditions d'arrêt

Lorsque la recherche tabou atteint 10000 itérations sans amélioration, elle s'arrête et demande au mécanisme de restart sur quelle solution elle doit reprendre.

4 Mécanisme de Restart

Nous avons implémenter un mécanisme de restart pour aider la recherche tabou à sortir des minimas locaux. Dans le suite on nommera *Cycle de recherche tabou* chaque cycle débutant au début d'une recherche tabou et finissant lorsque le nombre d'itérations sans amélioration atteint 10000. Ce mécanisme se déroule comme suit :

1. On commence avec une liste `Best_of_10` vide de taille maximale 10.
2. A chaque fin de *Cycle de recherche tabou* on rajoute la meilleure solution trouvée à droite de `Best_of_10`, si la liste est pleine, l'élément le plus à gauche est supprimé.
3. On trie `Best_of_10` de manière décroissante selon le coût des solutions afin de retirer la pire des meilleures solutions dans l'étape précédente. Puis on choisit une des 10 solutions de `Best_of_10` au hasard et on relance un *Cycle de recherche tabou* sur celle-ci en prenant soin de réinitialiser le `tabu_dict`.
4. Lorsque le temps alloué est écoulé ou que la borne inférieure de l'instance a été atteinte, on arrête l'algorithme.

5 Résultats & Discussion

5.1 Résultats comparatifs

Les algorithmes ont été exécutés sur un ordinateur doté d'un processeur i5 2.50GHz et 16GB de RAM en utilisant la graine 42. Nous allons comparer les résultats de l'algorithme globale (avec mécanisme de restart) avec ceux de sa version sans mécanisme de restart.

5.2 Évolution du score obtenu selon le nombre d'itération

Dans cette section nous présentons l'évolution du score obtenu en fonction du nombre d'itération effectué.

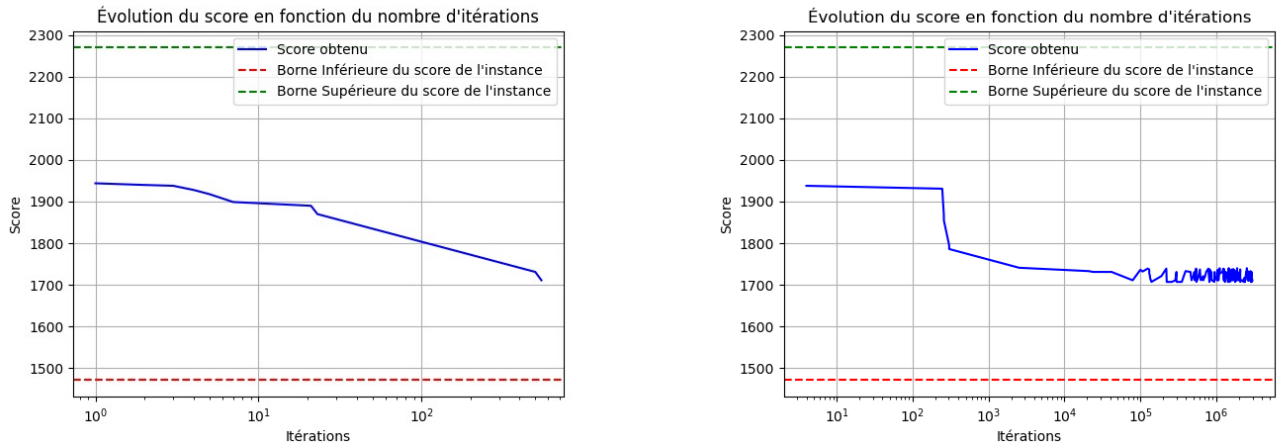


FIGURE 2 – Instance - A : Évolution du score obtenu en fonction du nombre d'itérations sans restart (à gauche) et avec restart (à droite).

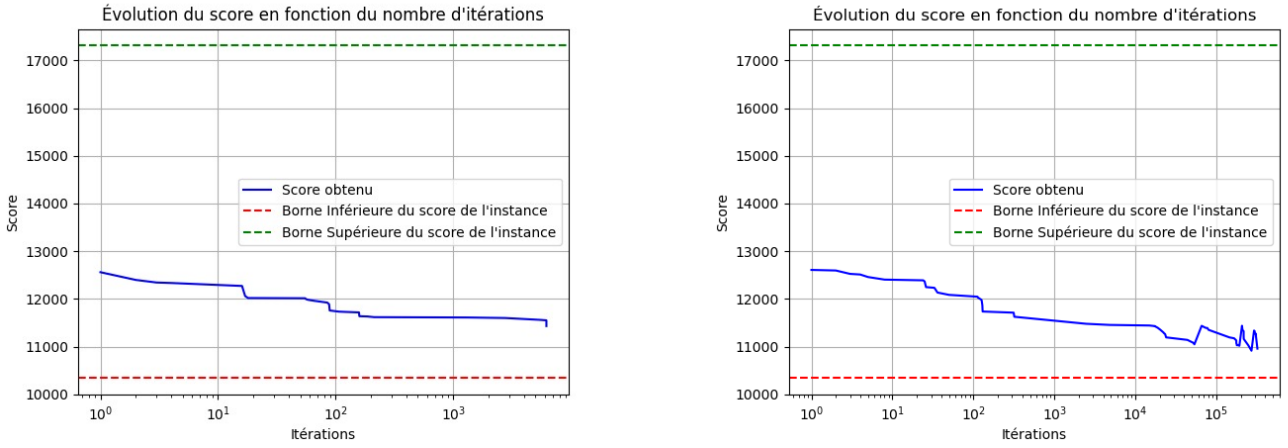
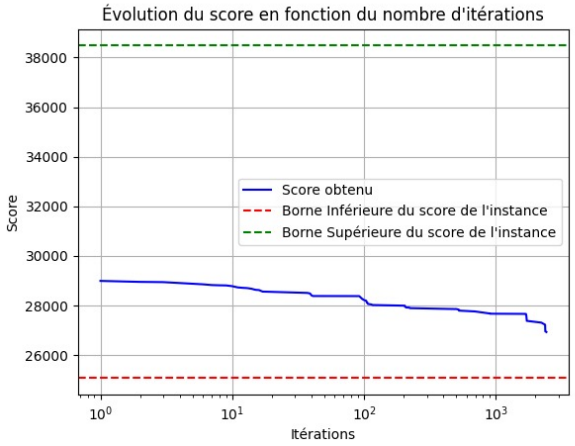


FIGURE 3 – Instance - B : Évolution du score obtenu en fonction du nombre d'itérations sans restart (à gauche) et avec restart (à droite).



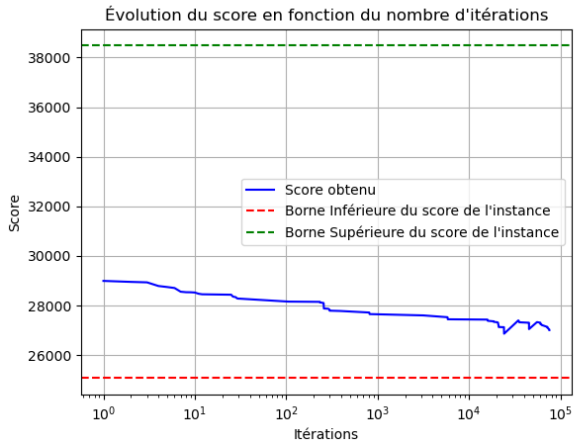
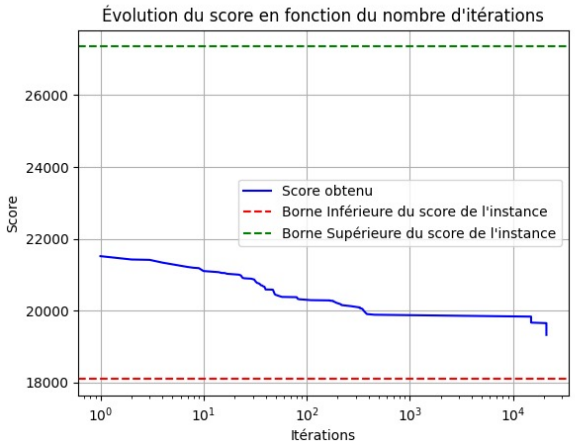


FIGURE 4 – Instance - C : Évolution du score obtenu en fonction du nombre d'itérations sans restart (à gauche) et avec restart (à droite).



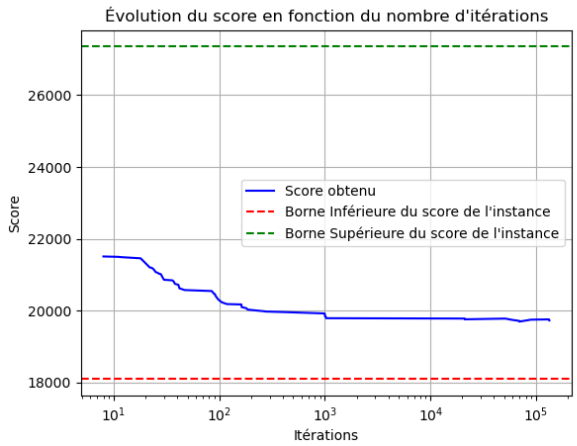
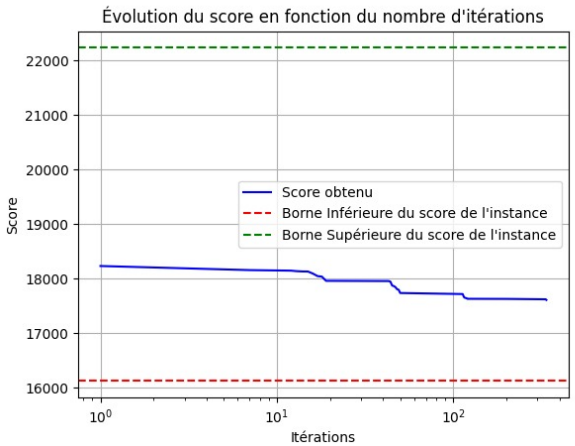


FIGURE 5 – Instance - D : Évolution du score obtenu en fonction du nombre d'itérations sans restart (à gauche) et avec restart (à droite).



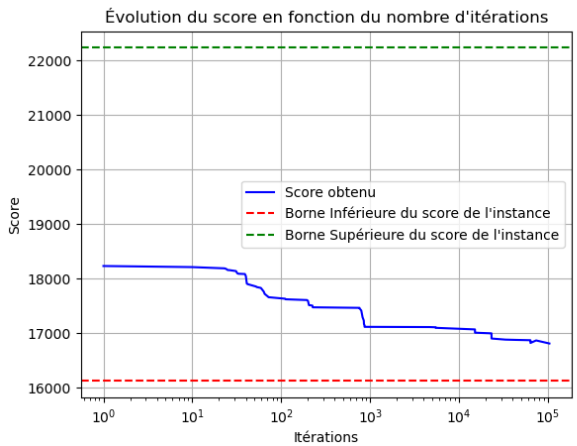


FIGURE 6 – Instance - E : Évolution du score obtenu en fonction du nombre d'itérations sans restart (à gauche) et avec restart (à droite).

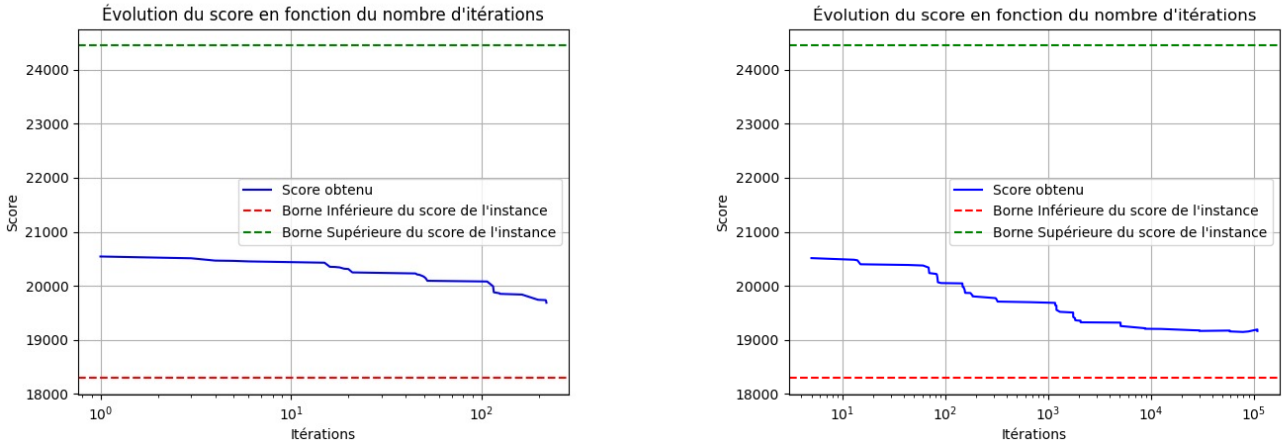


FIGURE 7 – Instance - F : Évolution du score obtenu en fonction du nombre d'itérations sans restart (à gauche) et avec restart (à droite).

Tout d'abord on remarque que le nombre d'itérations maximale bien plus faible pour l'algorithme sans restart. C'est cohérent dans la mesure où l'on affiche uniquement les itérations où des améliorations ont lieu sur ces graphiques. Dans le cas de l'algorithme sans restart, l'algorithme ne parvient plus à trouver d'amélioration a partir d'un certain nombre d'itérations. Dans le cas de l'algorithme avec restart, il parvient à en trouver d'autres car la meilleure solution "locale" à la recherche tabou change à chaque restart, ce qui explique la présence de zones à gradient positif dans les courbes de droite, absentes dans les courbes de gauche. La solution retournée en fin d'algorithme correspond à la meilleure solution trouvée au cours de la totalité de l'éxecution de l'algorithme.

Nous pouvons également voir que la méthode avec restart permet d'éviter des minimas locaux en trouvant de meilleures solutions jusqu'aux itérations de l'ordre de 10^5 alors que l'algorithme sans restart se retrouve rapidement bloquée dans les itérations de l'ordre de 10^2.

Les tableaux ci-dessous présentent les scores finaux obtenus selon les instances et les méthodes utilisées :

Instance	A	B	C	D	E	F
Meilleur score	1711	11431	26933	19328	17612	19686

TABLE 4 – Meilleur score sans restart pour les différentes instances.

Instance	A	B	C	D	E	F
Meilleur score	1707	10914	26866	19698	16814	19144

TABLE 5 – Meilleur score avec restart pour les différentes instances.

On remarque une amélioration de la qualité du score obtenu lors de l'utilisation du mécanisme de restart pour toutes les instances sauf pour la D. On voit donc que selon la nature de l'instance, l'intensification des solutions obtenues est plus importante que la diversification des solutions visitées. On remarque d'ailleurs très bien le plateau de plus de 10000 itérations sur le graphe gauche de la figure 5 avant de faire chuter brutalement le score.

5.3 Discussion

On discute ici de plusieurs améliorations ou pistes de réflexions possibles :

1. **Optimisation plus approfondie des hyperparamètres :** Dans un premier temps il serait intéressant de s'affranchir des compromis évoqué lors de la section 3.4 afin d'obtenir une optimisation des hyperparamètres plus fines.

Il s'agirait donc de reproduire ce qui a été fait pour l'instance D sur toutes les instances.

2. **Modification dynamique des hyperparamètres :** Une autre approche consisterait à modifier dynamiquement la valeurs des hyperparamètres. On peut nottament imaginer d'augmenter le pourcentage de voisins sélectionnés ou encore d'augmenter le temps de bannissement dans le `tabu_dict` d'une abstraction de solution lorsque l'on stagne. De la même manière on pourrait modifier dynamiquement le nombres d'itérations à effectuer avant de réaliser le mécanisme de restart.
3. **Choix de l'abstraction utilisée :** On pourrait également étudier différents choix d'abstraction des solutions, une abstraction plus forte ou plus faible impacterait forcément les performances de notre algorithme. Cette abstraction est celle qui nous semblait la plus pertinente mais nous n'avons pas eu le temps de tester d'autres abstractions.