POLYTECH LYON

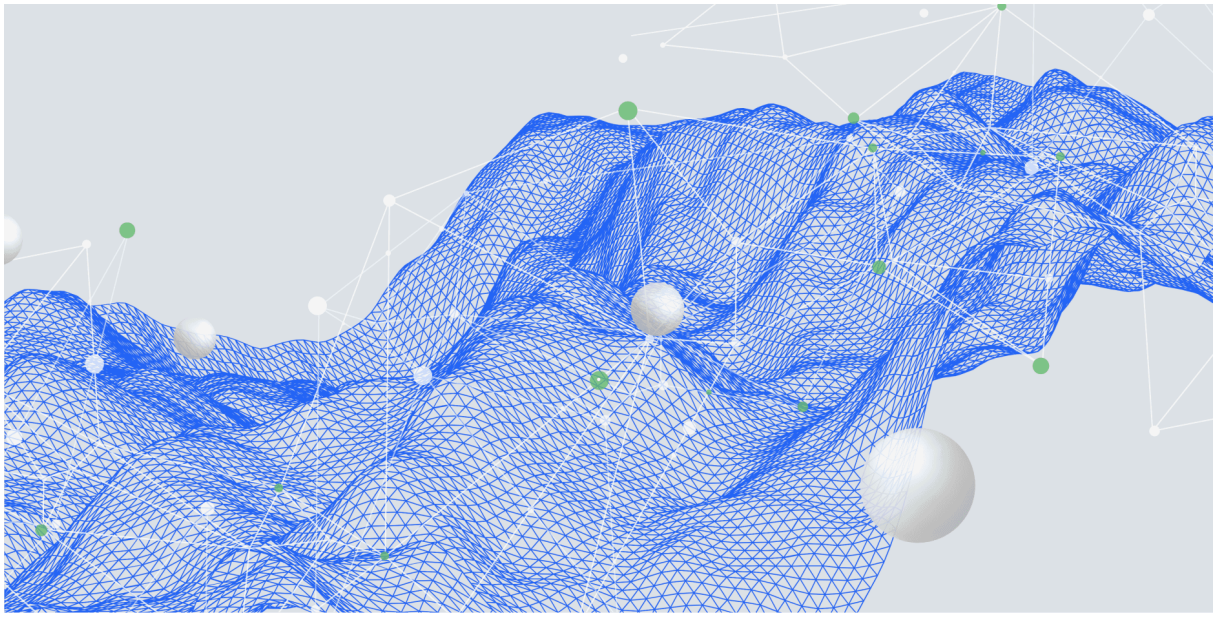
Lyon 1

Optimisations Discrètes

M. Stéphane Bonnevay

4A Informatique

Rapport Knapsack



Rédigé par
Amaury GALICHET
&
Cerine KERARMA

Rendu le 14 avril 2025

Sommaire

*le lien de notre repository git est en bibliographie

Introduction **2**
I. Solution optimale avec un solver **4**
II. Recuit simulé **5**
- Paramètres de l'algorithme 5
- Paramètres spécifiques des tests 5
- Processus de test 5
- Résultats des tests 6

Interprétations des résultats obtenus par nos tests sur le recuit simulé 12
III. Algorithme génétique **13**
- Paramètres de l'algorithme 13
- Paramètres spécifiques pour les tests 13
- Processus de test 14
- Résultats des tests 14

Interprétations des résultats obtenus par nos tests sur l'algo génétique 23
IV. Comparaison de temps d'exécution **24**
- ★ Pi-12 24
- ★ Pi-13 25
- ★ Pi-15 26

Conclusion **27**
Bibliographie **28**

Introduction

Dans ce rapport, nous abordons le problème du sac à dos et proposons une solution en utilisant des métaheuristiques dans le cadre du cours d'optimisations discrètes de 4eme année informatique à Polytech Lyon, encadré par M. Stéphane Bonnevay.

Le problème du sac à dos est un problème d'optimisation combinatoire où l'objectif est de maximiser la valeur totale d'objets que l'on peut mettre dans un sac à dos, tout en respectant une contrainte de poids. Plus précisément, on dispose de plusieurs objets, chacun ayant une valeur et un poids, et l'on doit décider quels objets emporter dans le sac à dos afin de maximiser la valeur totale, sans dépasser la capacité maximale du sac.

La solution ‘gloutonne’ serait de trier les objets par ordre décroissant de profit (O(n log n)) et de les ajouter un par un dans le sac à dos et dès qu’on atteint la borne inferieur du poids maximal, on ajoute une fraction de l’objet suivant dans la liste des profits afin de combler ce qui reste du poids manquant. Et on obtient donc notre solution avec un profit maximal tout en respectant le poids maximal. Cependant, la contrainte ici est que l'on est obligé d'utiliser l'entièreté de l’objet. C'est donc un problème 0/1 (binaire). La solution devient donc plus complexe car on est dans les entiers et non dans les réels.

Dans ce cas, on pourrait prendre toutes les combinaisons possibles avec des 0 et 1 pour n objets. Ensuite on choisit les solutions qui respectent le poids maximal et on sélectionne celle dont le profit est maximal. C'est la solution “naïve” qui peut bien évidemment fonctionner cependant elle est exponentielle en temps avec une complexité de O(2^n) ce qu'on veut absolument éviter.

Une méthode plus optimisée serait de décomposer notre problème en plusieurs sous-problèmes qui peuvent être résolus plus facilement et qui nous aideront petit a petit a trouver notre solution.

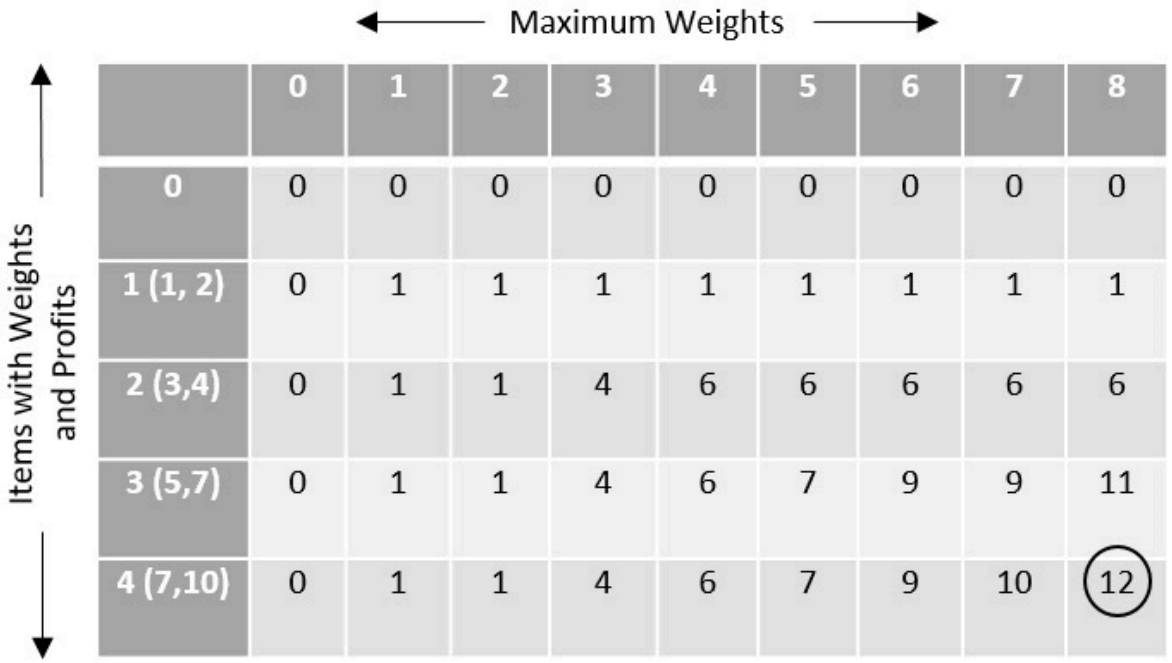
À petite échelle, une approche possible consiste à sélectionner un objet de manière aléatoire puis à vérifier qu'il ne dépasse pas la capacité restante du sac. Ensuite, on choisit uniquement les objets dont le poids ne dépasse pas le poids restant, c'est-à-dire la différence entre la capacité totale du sac et la somme des poids des objets déjà choisis. Cette procédure est répétée jusqu'à ce que la capacité maximale soit atteinte, ou que l’on ne puisse plus ajouter d’objet sans dépasser cette capacité maximale. Cette méthode est connue sous le nom de méthode de tabulation [1].

Avec la méthode de tabulation:

Poids maximal →	0	1	2	…	poids maximal défini
Objet 1 Poids: poids_1 Profit: profit_1	0	Profit_1 si poids_1 <= 1 sinon 0			
Objet 2 Poids: poids_2 Profit: profit_2					

…					
Objet n Poids: poids_n Profit: profit_n					Solution optimale

Tableau 1 : Tableau donné par la méthode de tabulation



	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (1, 2)	0	1	1	1	1	1	1	1	1
2 (3,4)	0	1	1	4	6	6	6	6	6
3 (5,7)	0	1	1	4	6	7	9	9	11
4 (7,10)	0	1	1	4	6	7	9	10	12

Figure 1 : Exemple concret de l'implémentation de la méthode de tabulation sur un problème de sac à dos simple

(source: Knapsack Problem in Data Structures)

Le but est de diviser notre problème en sous-problèmes faciles à résoudre. Pour chaque item on vérifie si son poids est inférieur au poids maximal (0,1,2,3,...,poids maximal du problème) et on prend la valeur du profit si l'objet respecte le poids posé. S'il existe un item dans la ligne précédente qui peut être sélectionné, on prend son profit afin d'avoir toujours le plus grand profit. Le but est donc de trouver le profit maximal qui respecte le poids maximal en décomposant le problème en sous problèmes. Le profit maximal se trouverait à la dernière case de la dernière ligne et il suffit de revenir en arrière (par récursivité) pour trouver les objets qu'on a choisi pour arriver à cette solution optimale avec une complexité de O(n*m) ou m est le poids maximal et n le nombre d'objets total.

Cette méthode est optimale lorsqu'on dispose de peu de données, c'est-à-dire pour des tableaux de petite taille. Cependant, dès que la taille des données devient plus importante, par exemple de l'ordre de 100 éléments ou plus, la récursivité devient une technique moins adaptée. En effet, on risque de dépasser le nombre d'appels récursifs autorisé, ce qui peut entraîner une erreur de dépassement de pile, ou pire encore, on risque d'attendre une éternité pour obtenir un résultat.

C'est pourquoi, dans ces cas-là, on se tourne vers les métaheuristiques. Ces méthodes n'assurent pas nécessairement l'obtention de la solution optimale mais elles permettent d'approcher de manière efficace une solution optimale même pour des problèmes de grande taille.

Les deux métaheuristiques que nous avons choisies d'explorer sont : le recuit simulé, basé sur la méthode de voisinage, et l'algorithme génétique, qui utilise la génération de populations.

I. Solution optimale avec un solver

Avant de commencer les tests des métaheuristiques qu'on a décidé d'implémenter, on a utilisé un solver exact pour trouver la solution optimale du problème du sac à dos ainsi que les valeurs associées au poids et au profit maximal. Cela nous a permis de comparer les résultats obtenus par les algorithmes métaheuristiques à une référence absolue.

Nous avons trouvé la solution optimale sur trois fichiers différents (pi12, pi13, et pi15), chacun correspondant à un ensemble d'items avec différentes tailles, à savoir n=100, n=1000 et n=10 000.

Voici les données que nous avons obtenues à l'aide du solver:

Nom du fichier	Taille des données	Profit maximal	Poids à ne pas dépasser	Poids atteint	Nb d'objets ajoutés dans le sac à dos	Temps d'exécution (en s)
Pi 12	100	970	970	970	1	0.014
	1000	4514	4556	4528	6	0.073
	10 000	45 105	45 132	45 105	101	6.394
Pi 13	100	1989	970	969	17	0.010
	1000	6513	3177	3173	17	0.038
	10 000	64 077	31 234	31 217	278	4.289
Pi 15	100	1011	997	997	8	0.017
	1000	4950	4816	4816	77	0.093
	10 000	50 622	49297	49297	794	3.535

Tableau 2 : Résultats des solutions optimales donnés par un solveur exact

II. Recuit simulé

Le recuit simulé (RS) est une méthode d'optimisation inspirée du processus de refroidissement des métaux. L'algorithme explore les solutions en effectuant des mouvements à travers l'espace de recherche, acceptant parfois des solutions moins bonnes afin d'éviter de se retrouver dans des minima locaux. L'objectif est de s'approcher d'une solution optimale en explorant diverses configurations de manière itérative.

- **Paramètres de l'algorithme**

Température (T) : Ce paramètre détermine la probabilité d'accepter une solution moins bonne. Il commence à une valeur élevée et décroît progressivement au fur et à mesure des itérations. Plus la température est haute, plus la probabilité d'accepter une solution sous-optimale est grande.

Facteur de refroidissement (p0) : Ce facteur définit la vitesse à laquelle la température décroît à chaque itération. Un facteur de refroidissement proche de 1 signifie un refroidissement lent, permettant une exploration plus approfondie, tandis qu'un facteur proche de 0 signifie un refroidissement rapide.

- **Paramètres spécifiques des tests**

Nous avons testé cet algorithme sur trois fichiers différents (pi12, pi13, et pi15), chacun correspondant à un ensemble d'items avec différentes tailles, à savoir n=100, n=1000 et n=10 000.

Les valeurs de la température (T) choisies pour nos tests sont les suivantes :
100, 200, 500, 700, 1000, 2000, 5000, 7000, 10000.

Les valeurs des facteurs de refroidissement (p0) choisies sont :
0.85, 0.86, 0.87, ..., 0.99.

- **Processus de test**

On génère une solution initiale aléatoire pour chaque couple de paramètres (T, p0) tout en respectant la contrainte du poids. Pour chaque test, on a collecté les résultats dans un tableau (fichier csv) comprenant les informations suivantes :

- La solution initiale générée
- Le poids et profit associés à la solution initiale,
- Les paramètres (T et p0),
- La nouvelle solution après l'application de l'algorithme,
- Les nouveaux profits et poids obtenus,
- Le nombre d'itérations nécessaires pour atteindre la solution finale,
- Le temps d'exécution de l'algorithme.

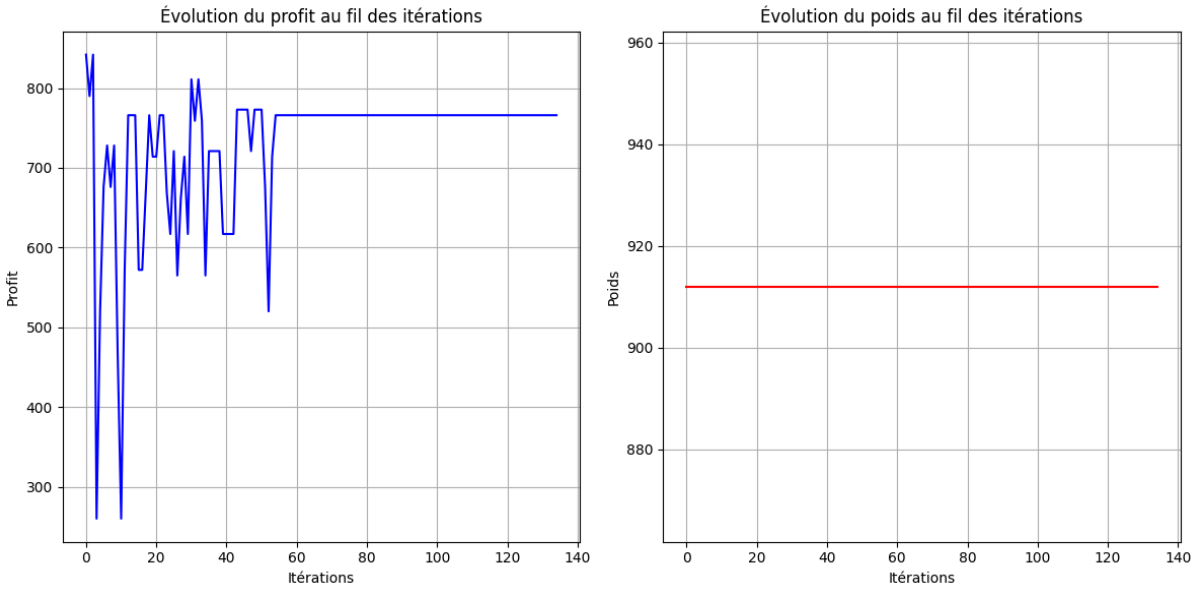
- **Résultats des tests**

On a réalisé les tests sur tous les fichiers (pi-12, pi-13 et pi-15) sur toutes les tailles de données (n=100, 1000 et 10 000). Ces tests nous ont permis de tracer des graphiques afin d'illustrer les valeurs du profit et du poids obtenus en fonction des paramètres choisis. Plus précisément, ces graphiques présentent l'évolution du poids minimal et du profit maximal en fonction de la température initiale T0 et du facteur de refroidissement p.

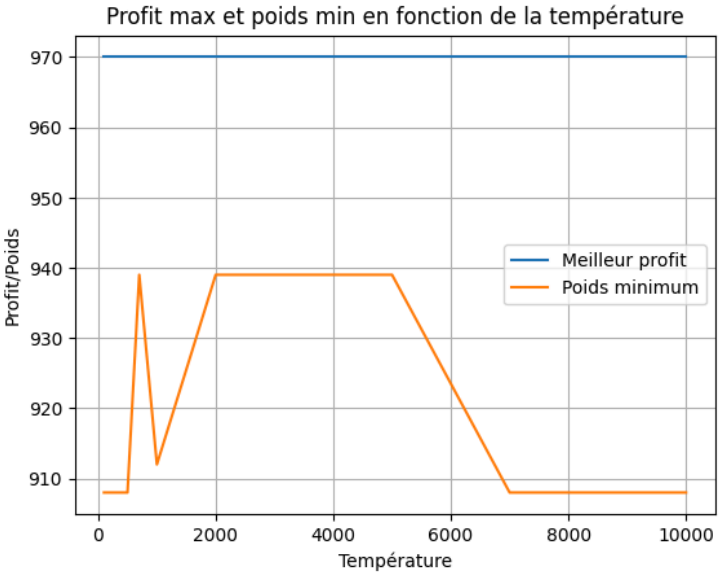
Voici les résultats obtenus:

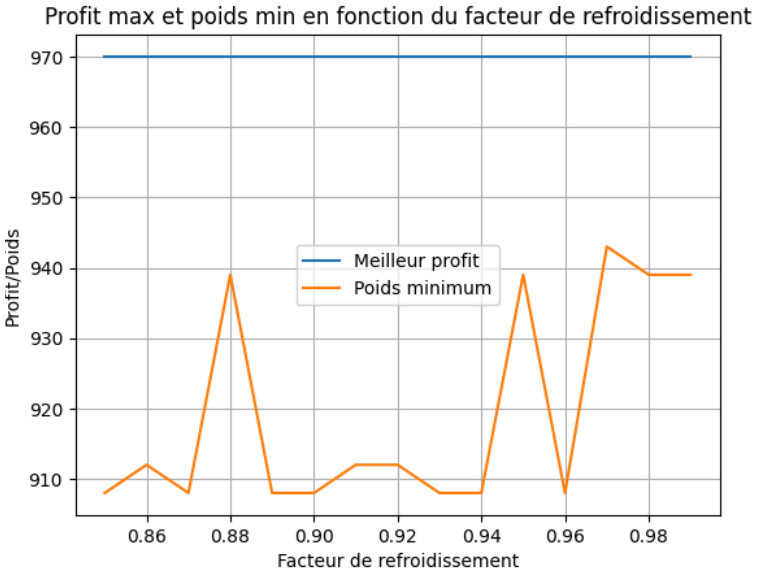
★ RÉSULTATS PI-12:
- ❖ n=100

Test du RS avec des paramètres fixés aléatoirement pour suivre l'évolution de l'algo au fil des itérations. Ce graphique montre l'évolution de la solution pour une température et facteur refroidissement précis. *Pour trouver davantage de graphiques avec des paramètres fixés, voir les instructions dans le README.md dans notre repo git [2].*



Nous avons ensuite tracé les graphiques du poids minimal et profit maximal en fonction de chaque paramètre:





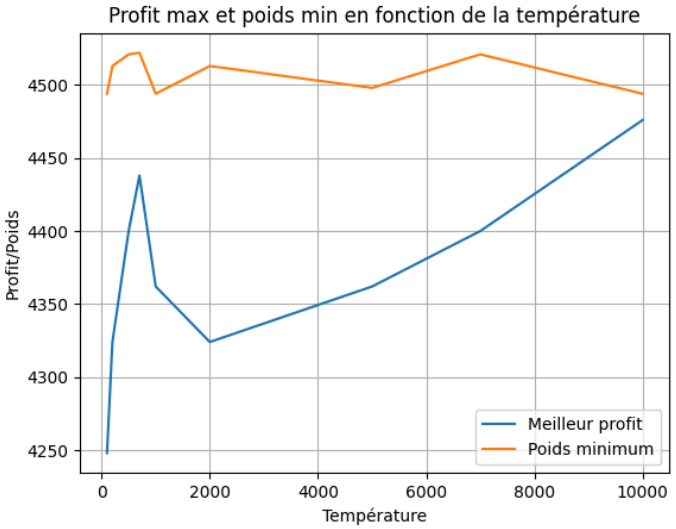
- Analyse statistique:

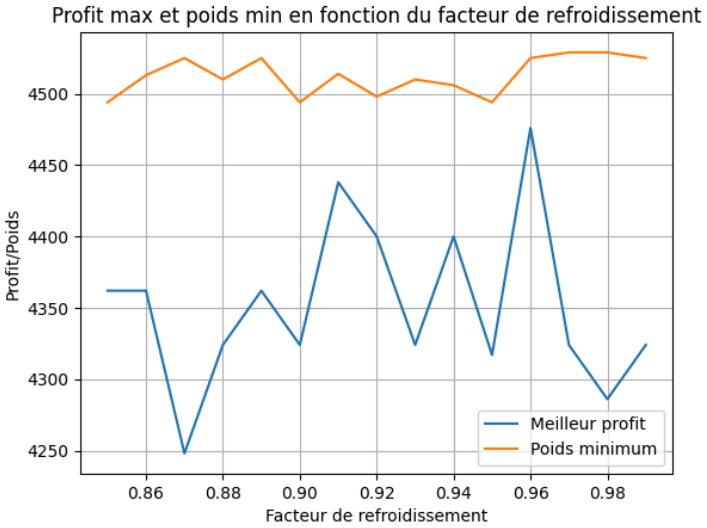
On a procédé à une analyse approfondie des résultats en étudiant chaque fichier individuellement afin de vérifier si la solution finale fournie par le RS correspondait à la solution optimale (c'est-à-dire celle trouvée par le solver) en termes de poids et de profit.

Plus précisément, pour le fichier pi12 avec n=100 objets, on a réalisé une analyse statistique afin d'identifier la configuration de paramètres ayant permis, le plus souvent, d'atteindre la solution optimale. Après traitement et observation des résultats, on a pu déterminer les valeurs suivantes comme étant les paramètres optimaux pour ce cas :

➢ La température optimale est 200
➢ Le facteur de refroidissement optimal est 0.93

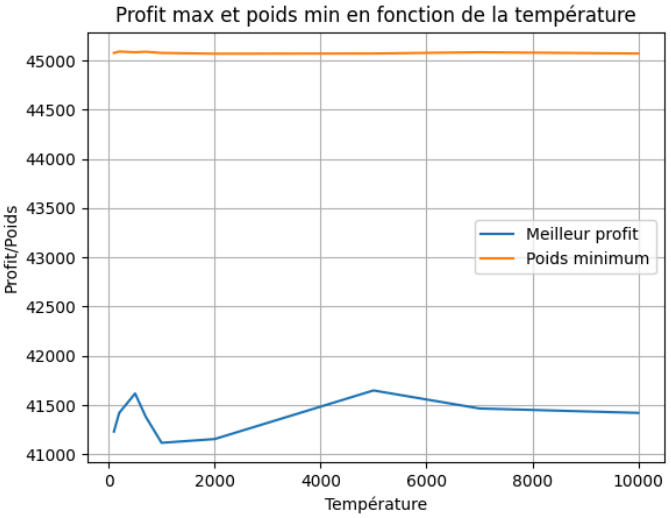
❖ n=1000

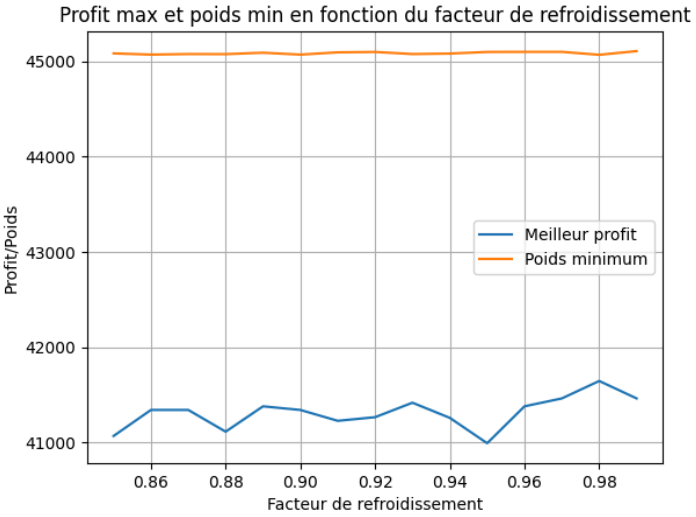




Pour le fichier pi12 avec n=1000, nous n'avons pas trouvé de solutions qui atteignent le même profit et poids que la solution donnée par le solver.

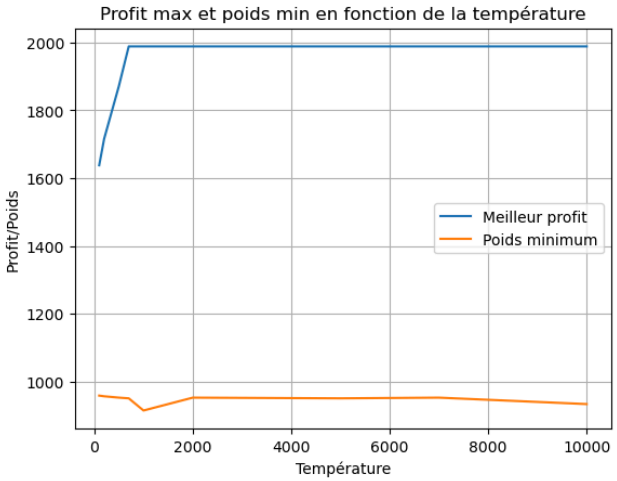
- ❖ n=10 000

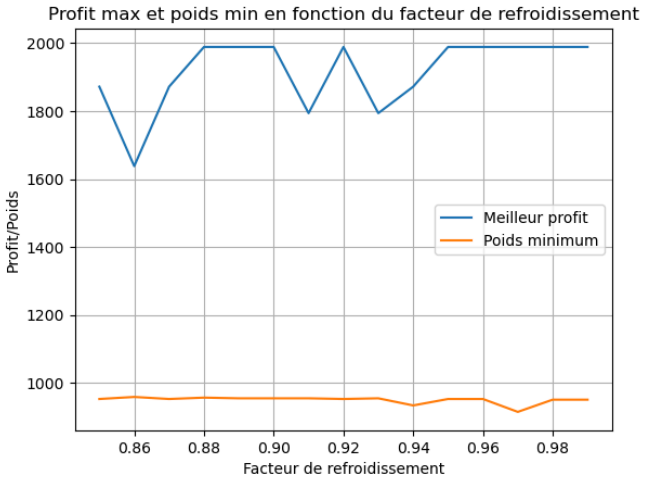




Pour le fichier pi12 avec n=10 000, nous n'avons pas trouvé de solutions qui atteignent le même profit et poids que la solution donnée par le solver.

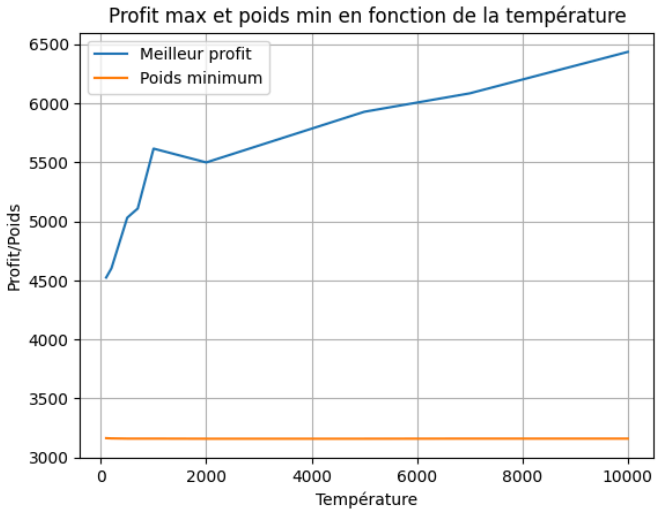
- ★ RÉSULTATS PI-13:
  - ❖ n=100

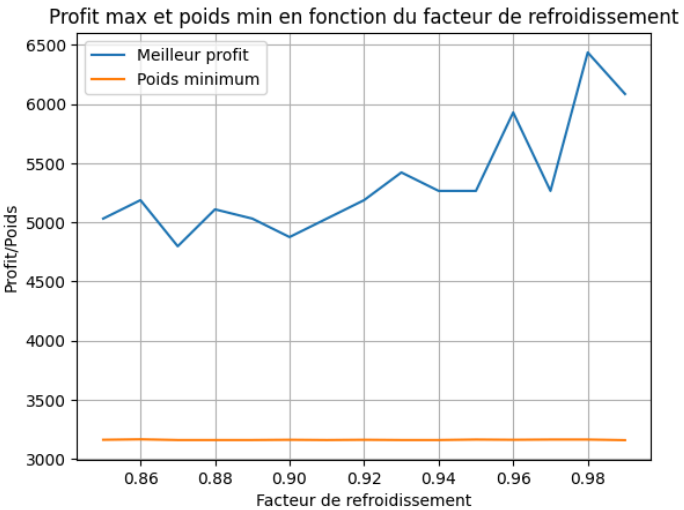




Pour le fichier pi13 avec n=100, nous n'avons pas trouvé de solutions qui atteignent le même profit et poids que la solution donnée par le solver.

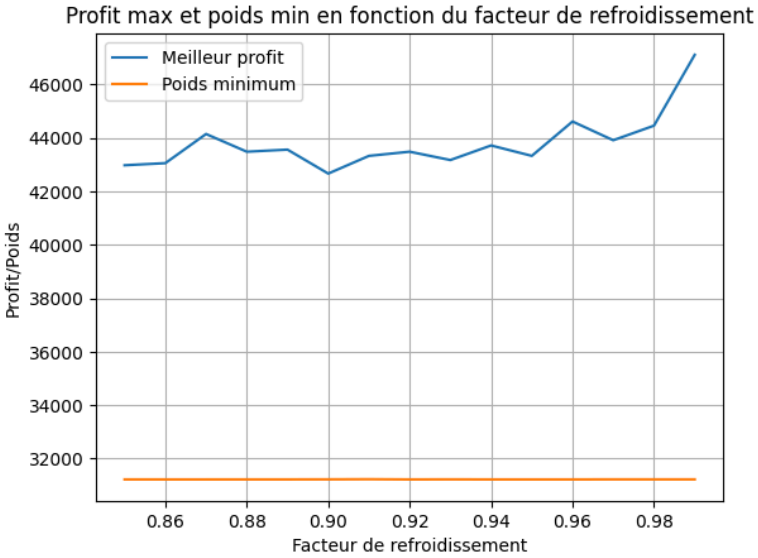
- n=1000

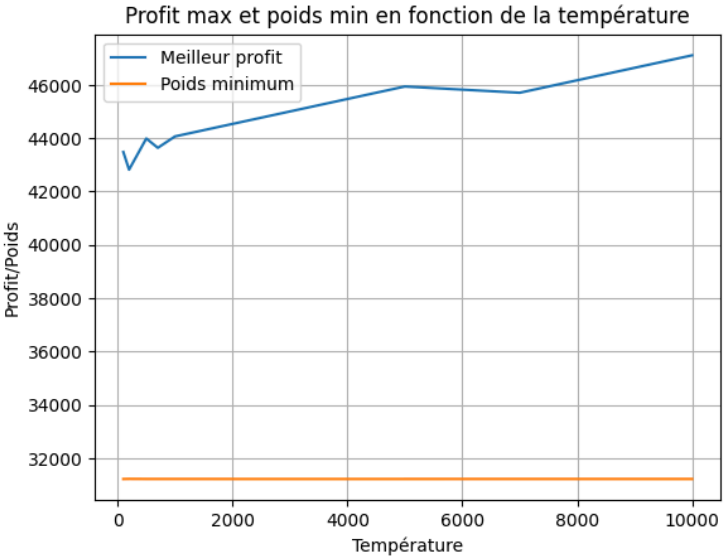




Pour le fichier pi13 avec n=1000, nous n'avons pas trouvé de solutions qui atteignent le même profit et poids que la solution donnée par le solver.

- n=10 000

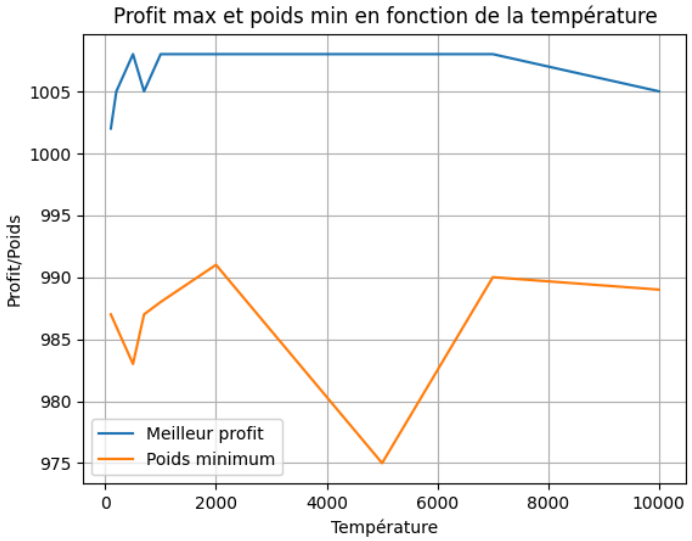


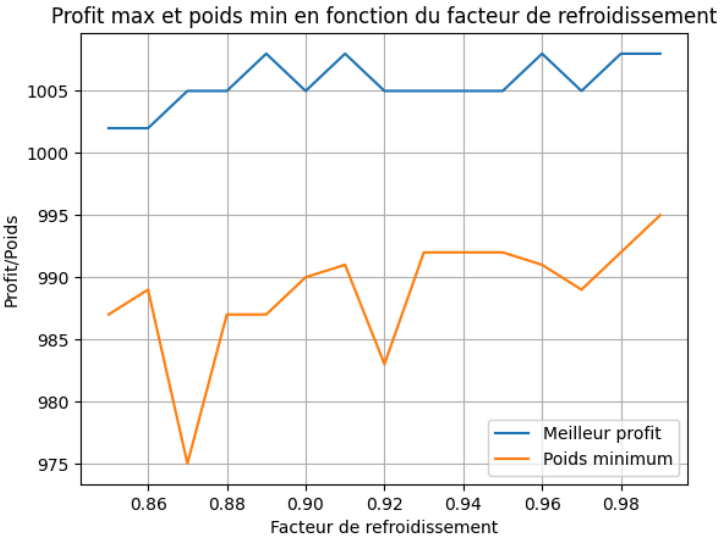


Pour le fichier pi13 avec n=10000, nous n'avons pas trouvé de solutions qui atteignent le même profit et poids que la solution donnée par le solver.

★ RÉSULTATS PI-15:

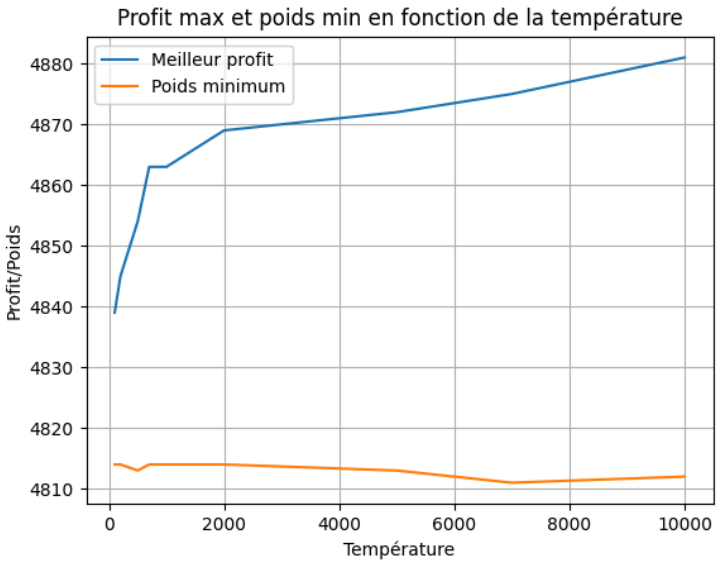
- n=100

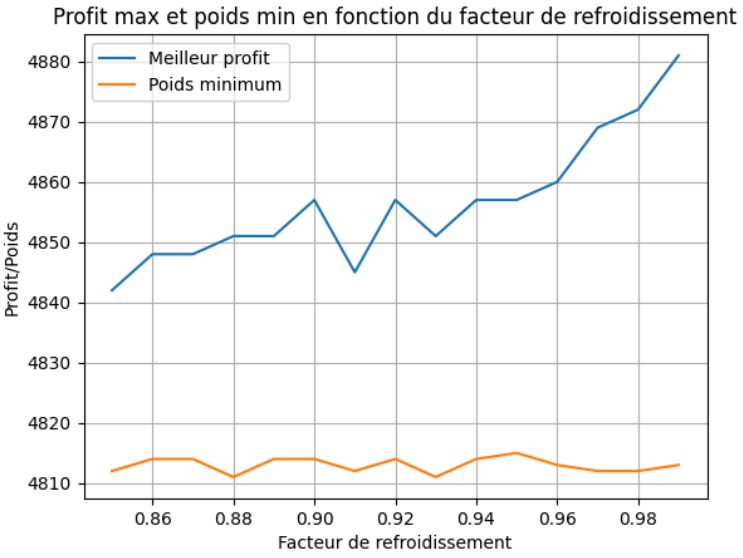




Pour le fichier pi15 avec n=100, nous n'avons pas trouvé de solutions qui atteignent le même profit et poids que la solution donnée par le solver.

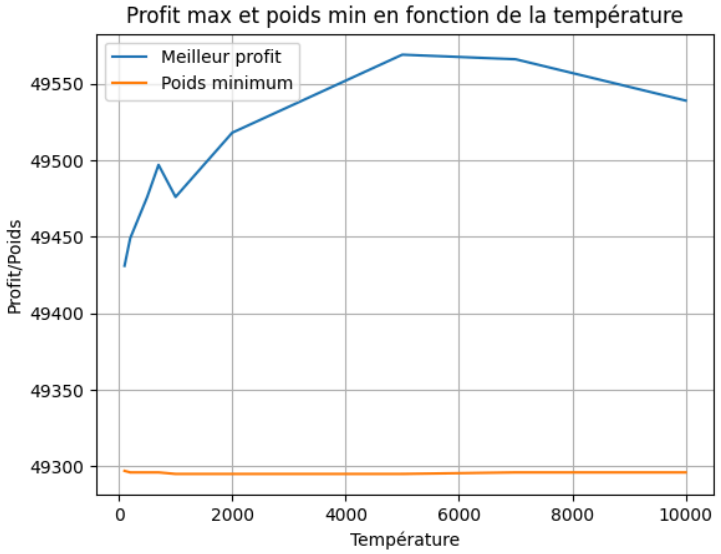
❖ n=1000

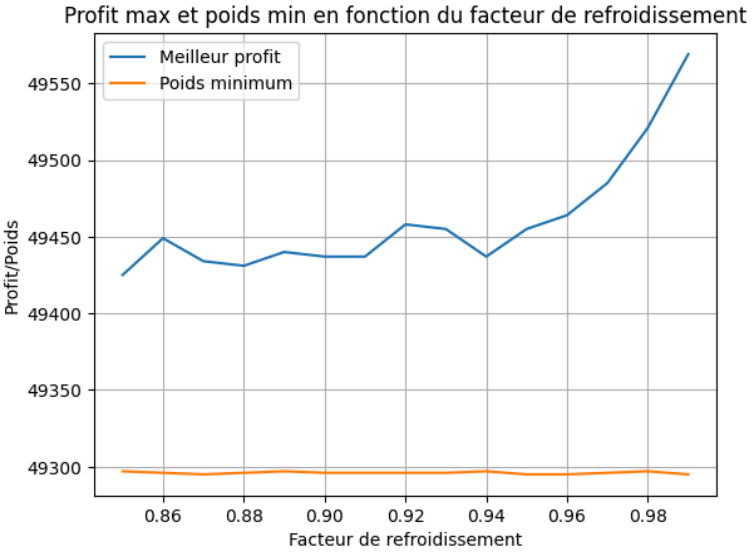




Pour le fichier pi15 avec n=1000, nous n'avons pas trouvé de solutions qui atteignent le même profit et poids que la solution donnée par le solver.

❖ n=10 000





Pour le fichier pi15 avec n=10 000, nous n'avons pas trouvé de solutions qui atteignent le même profit et poids que la solution donnée par le solver.

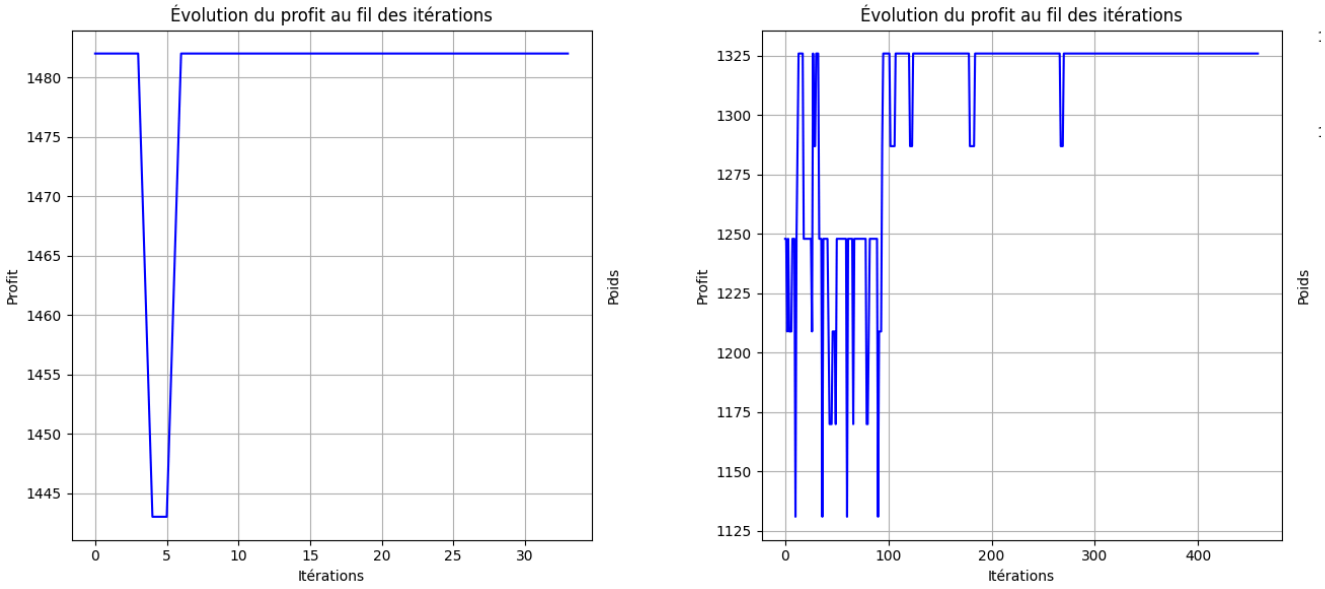
Interprétations des résultats obtenus par nos tests sur le recuit simulé

On constate que l'algorithme de recuit simulé parvient systématiquement à fournir une solution sans jamais rester bloqué. La seule fois où notre algorithme a donné exactement le même profit et le même poids que la solution optimale absolue a été pour le fichier pi12 avec n=100 avec une température initiale de 200 et un facteur de refroidissement de 0.93. Aucune autre configuration n'a permis d'atteindre à la fois le profit maximal et le poids minimal.

Cependant, il est important de noter que le profit maximal a été atteint à plusieurs reprises par les solutions générées mais avec un poids supérieur à celui de la solution optimale. On considère ces solutions comme bonnes dans la mesure où le critère principal de notre fonction objectif est la maximisation du profit. Néanmoins, en pratique, il serait préférable de minimiser également le poids total afin de rendre le sac à dos le plus léger possible ce qui n'a pas été pris en compte dans notre fonction objectif.

En outre, on a remarqué que seules les solutions de taille n=100 ont permis d'atteindre le profit maximal. Cela peut s'expliquer par le fait que, pour des tailles plus grandes, l'espace de solutions est beaucoup plus vaste, avec un nombre de voisins considérablement plus important, rendant l'exploration exhaustive plus difficile. L'algorithme finit alors par explorer uniquement une partie restreinte de l'espace ce qui peut expliquer pourquoi il n'atteind pas le profit optimal pour ces cas.

Enfin, les tests montrent que pour une température initiale donnée, plus le facteur de refroidissement est élevé, plus l'algorithme explore de solutions.
Par exemple, pour le fichier pi12 avec n=100 et une température initiale de 100 avec un facteur p=0.87, le nombre de solutions explorées reste limité alors qu'avec p=0.99, l'algorithme explore un espace bien plus large:



Évolution du profit au fil des itérations pour T0=100 et p=0.87 (a gauche) puis p=0.99 (à droite)

Cela montre que le facteur de refroidissement joue un rôle crucial dans la capacité de l'algorithme à explorer efficacement l'espace des solutions.

D'après nos observations et les graphiques obtenus, on conclut donc que Plus T_0 est élevé, plus l'algorithme accepte des solutions moins bonnes au début ce qui permet une meilleure exploration de l'espace des solutions. Si T_0 est trop bas, l'algorithme reste coincé dans un minimum local et ne trouve pas une bonne solution.

Quant à p, il contrôle la vitesse à laquelle la température diminue. Donc si p est proche de 1, le refroidissement est lent ce qui donne une meilleure exploration mais est plus lent à converger.

Si p est plus petit, le refroidissement est rapide ce qui signifie qu'on convergera rapidement vers une solution qui pourrait ne pas être optimale.

III. Algorithme génétique

L'algorithme génétique (AG) est une méthode de recherche inspirée des mécanismes de l'évolution naturelle tels que la sélection, la mutation, le croisement et la reproduction. Il permet de trouver des solutions optimales ou proches de l'optimal pour des problèmes complexes souvent en combinant des solutions de manière itérative.

- **Paramètres de l'algorithme**

NbPop : Taille de la population, c'est-à-dire le nombre de solutions candidates générées à chaque génération.

NbGen : Nombre de générations, soit le nombre de cycles d'itération que l'algorithme exécutera pour améliorer les solutions.

ProbaCross : Probabilité de croisement, qui détermine la probabilité que deux solutions parentales se croisent pour produire une solution enfant.

NbBest : Nombre de meilleures solutions conservées, c'est-à-dire le nombre de solutions les plus performantes qui sont conservées d'une génération à l'autre.

- **Paramètres spécifiques pour les tests**

Les tests sont réalisés avec les valeurs suivantes pour chaque paramètre :

NbPop_list = 10, 50, 70, 100
NbGen_list = 10, 50, 70, 100
ProbaCross_list = 0.1, 0.5, 0.9
NbBest_list = 2, 5

- **Processus de test**

On génère une solution initiale aléatoire pour chaque groupe de paramètres (NbPop, NbGen, ProbaCross, et NbBest) tout en respectant la contrainte du poids. Pour chaque test, on collecte les résultats dans un tableau (fichier csv) comprenant les informations suivantes:

- La solution initiale générée
- Le poids et profit associés à la solution initiale,
- Les paramètres (NbPop, NbGen, ProbaCross et NbBest),
- La nouvelle solution après l'application de l'algorithme,
- Les nouveaux profits et poids obtenus,
- Le nombre d'objets ajoutés,
- Le temps d'exécution de l'algorithme.

- **Résultats des tests**

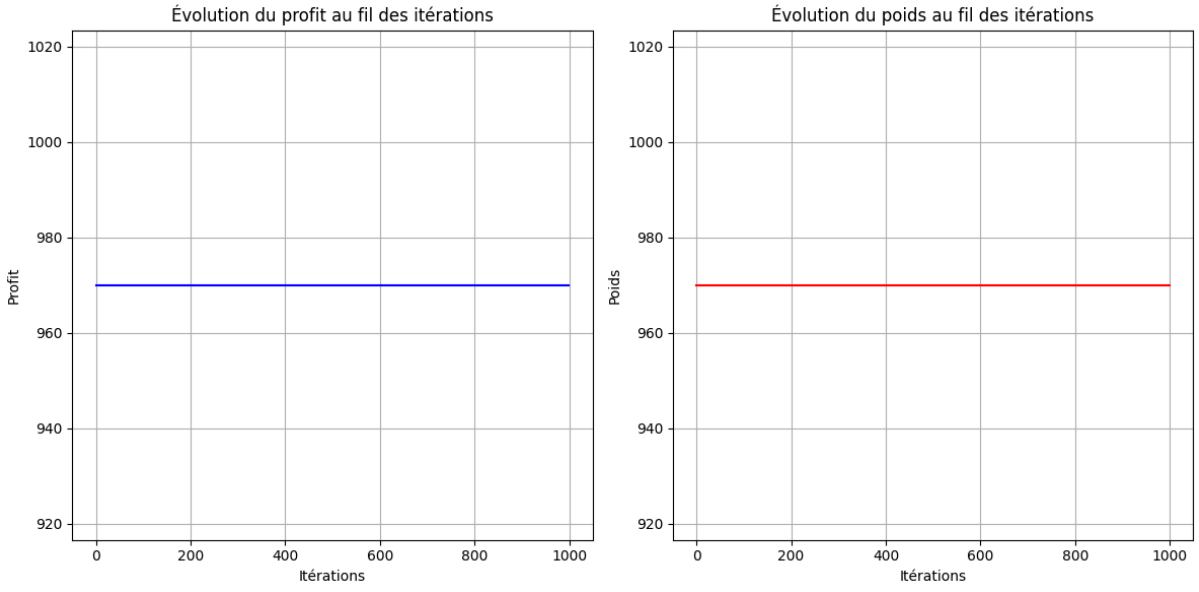
Les tests ont été réalisés sur tous les fichiers et toutes les tailles de données disponibles. À partir des résultats obtenus, des graphiques ont été tracés pour observer l'impact de chaque paramètre (tels que NbPop, NbGen, ProbaCross, et NbBest) sur le poids minimal et le profit maximal.

Voici les résultats obtenus:

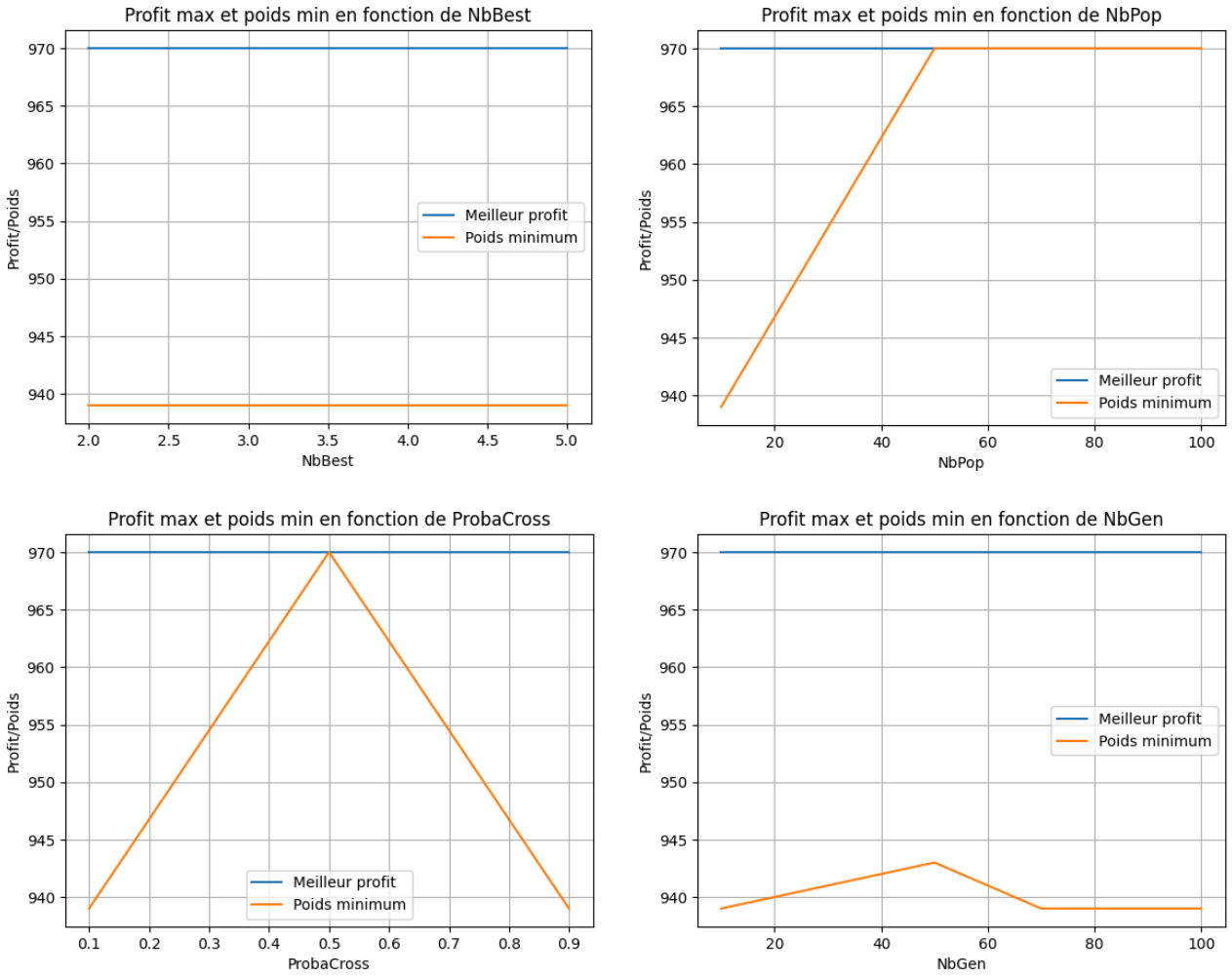
★ RÉSULTATS PI-12:

❖ n=100

Test de l'AG avec des paramètres fixés aléatoirement pour suivre l'évolution de l'algo au fil des générations. *Pour trouver davantage de graphiques avec des paramètres fixés, voir les instructions dans le README.md dans notre repo git [2].*



Graphiques illustrant le poids minimal et le profit maximal en fonction de chaque paramètre:



- Analyse statistique:

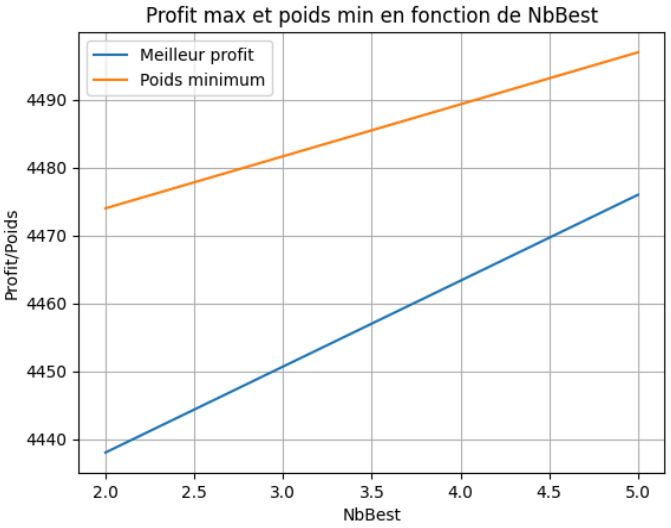
Ensuite, on a procédé à une analyse approfondie des résultats en étudiant chaque fichier individuellement afin de vérifier si la solution finale fournie par l'AG correspondait à la solution optimale (c'est-à-dire celle trouvée par le solver) en termes de poids et de profit.

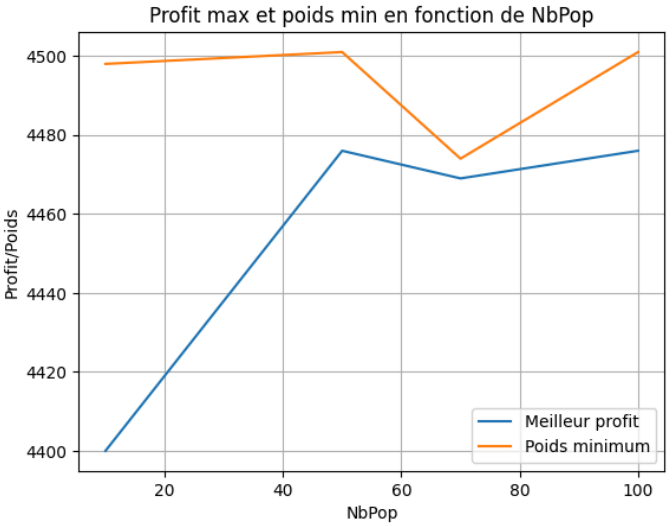
Plus précisément, pour le fichier pi12 avec n = 100 objets, on a réalisé une analyse statistique afin d'identifier la configuration de paramètres ayant permis, le plus souvent, d'atteindre la solution optimale. Après traitement et observation des résultats, on a pu déterminer les valeurs suivantes comme étant les paramètres optimaux pour ce cas :

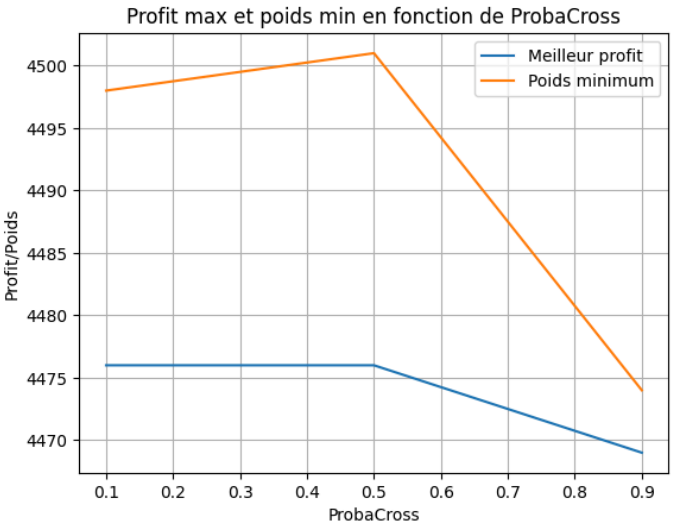
➢ Taille de la population (NbPop) : 100
➢ Nombre de générations (NbGen) : 100
➢ Probabilité de croisement (ProbaCross) : 0.9
➢ Nombre de meilleures solutions conservées (NbBest) : 2

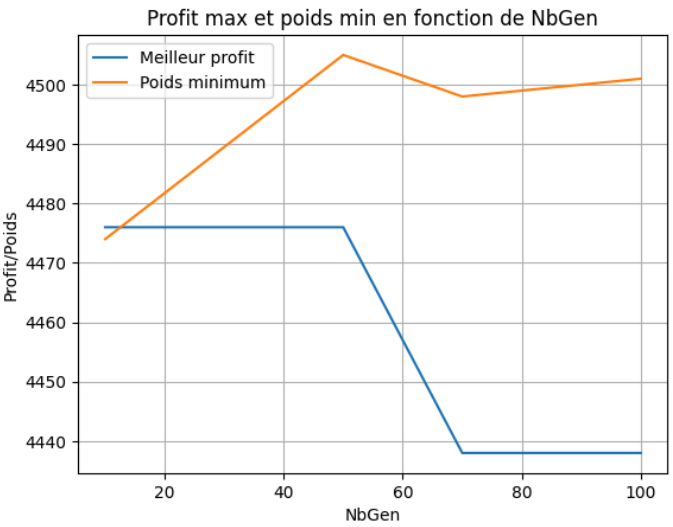
❖ n=1000

Graphiques





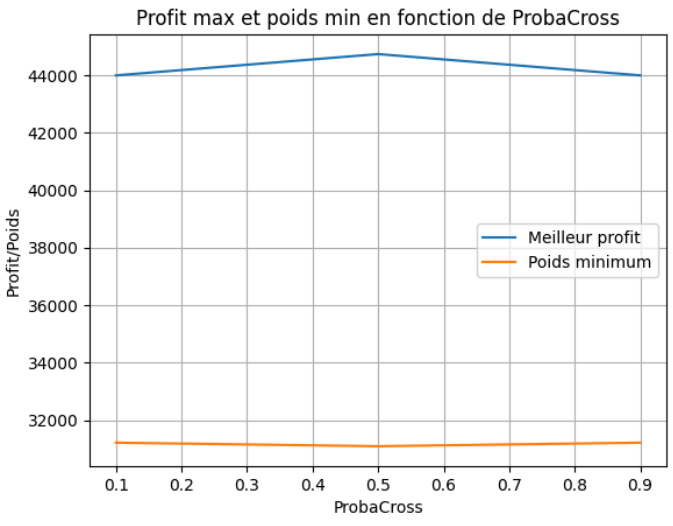


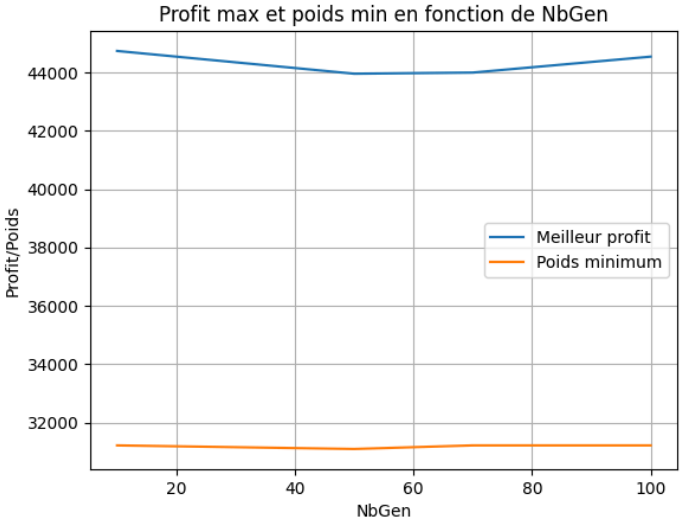


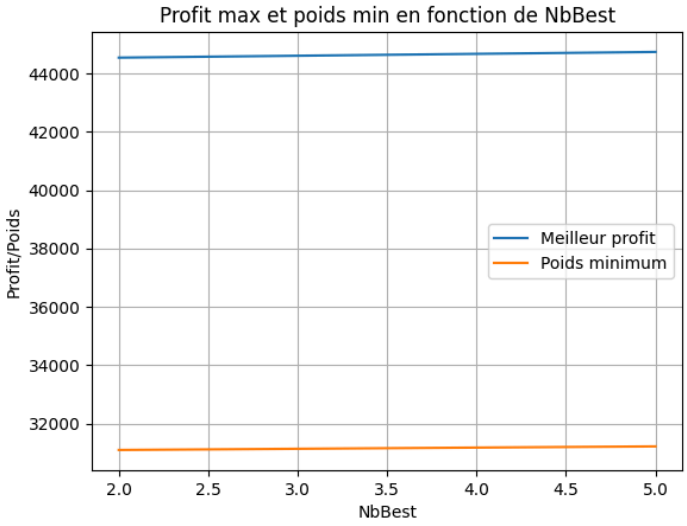
Pour le fichier pi12 avec n=1000, nous n'avons pas trouvé de solutions qui atteignent le même profit et poids que la solution donnée par le solver.

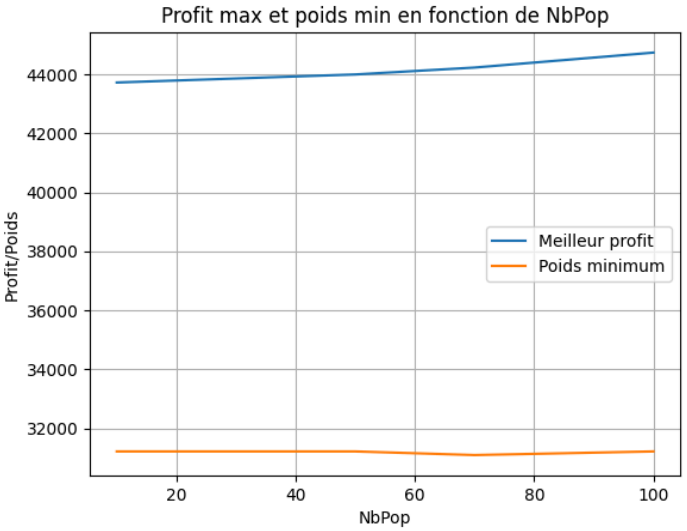
- n=10 000

Graphiques





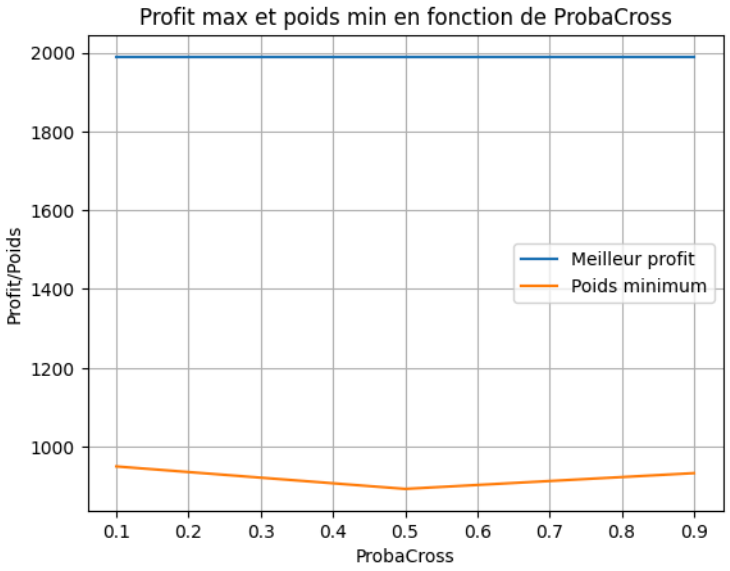


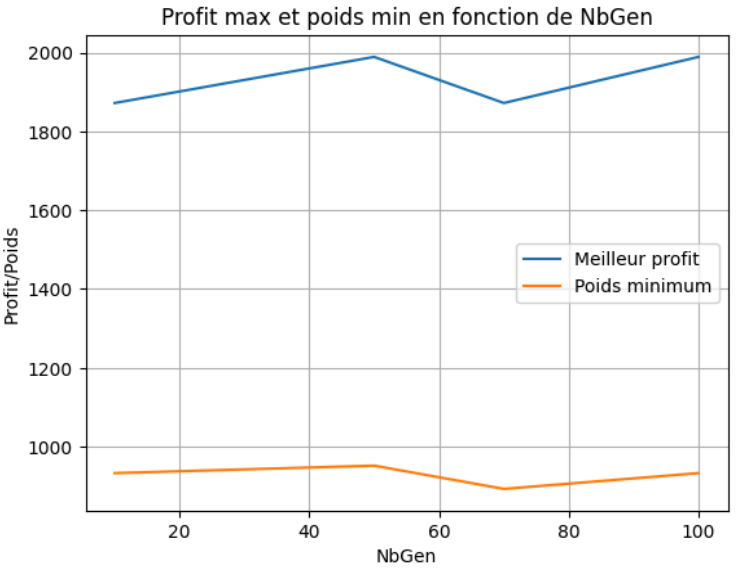


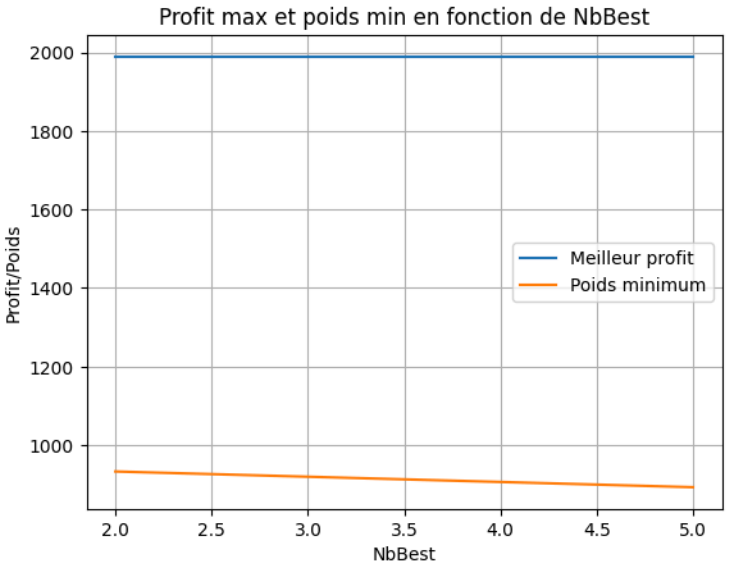
Pour le fichier pi12 avec n=10 000, nous n'avons pas trouvé de solutions qui atteignent le même profit et poids que la solution donnée par le solver.

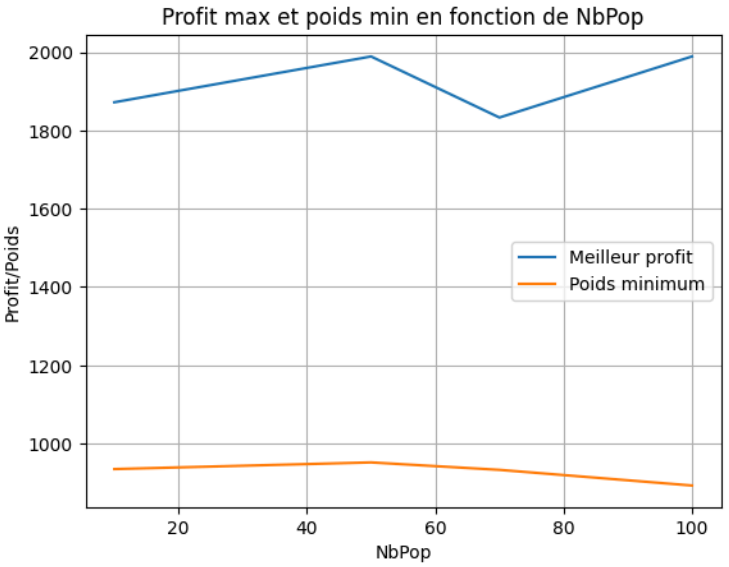
★ RÉSULTATS PI-13:

- ❖ n=100







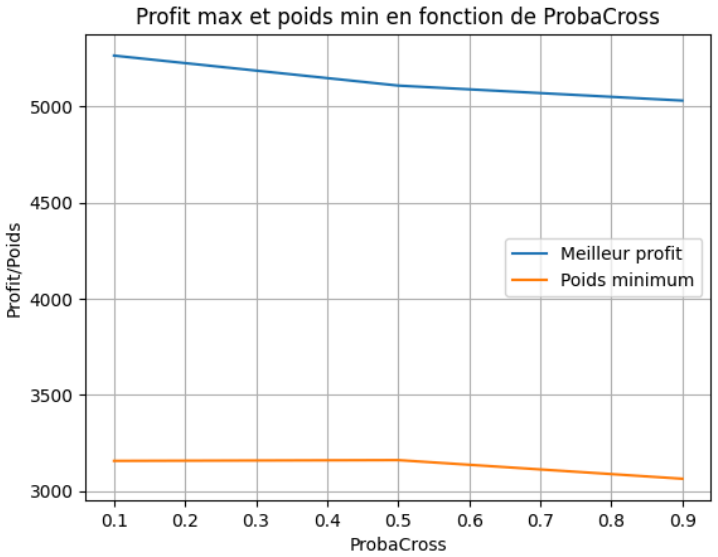


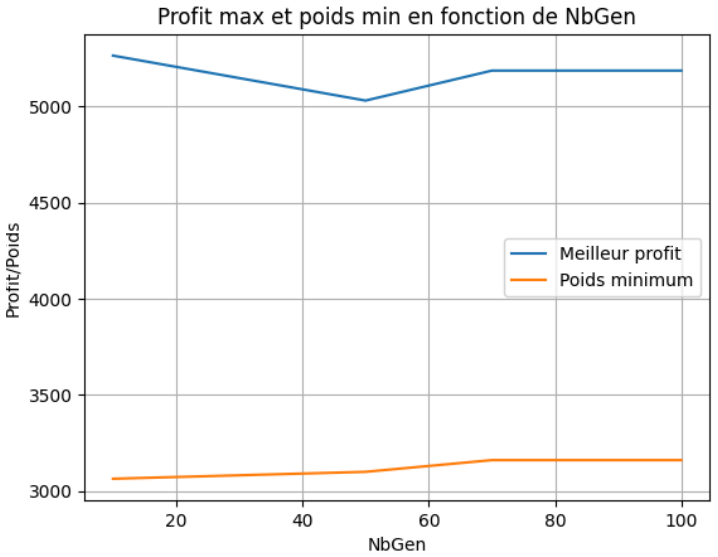
- Analyse statistique:

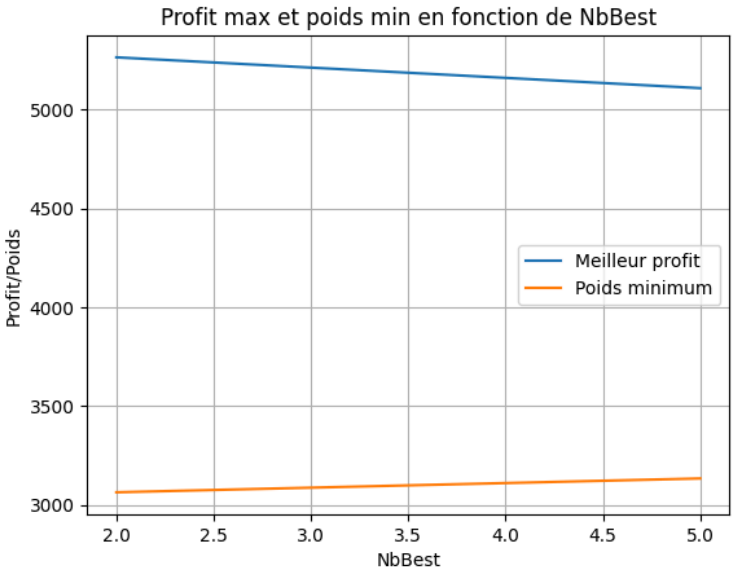
Le nombre de population optimal est: 50
Le nombre de génération optimal est: 100
La probabilité de croisement optimale est: 0.5
Le nombre de meilleures solutions optimal est: 5

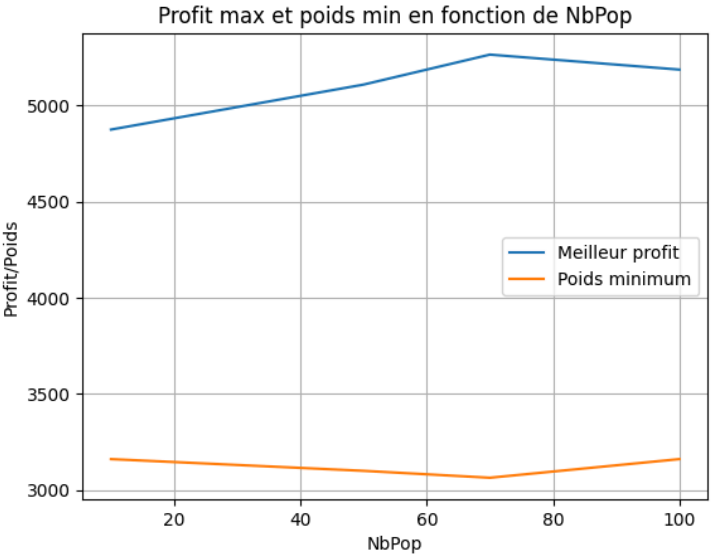
❖ n=1000

Graphiques



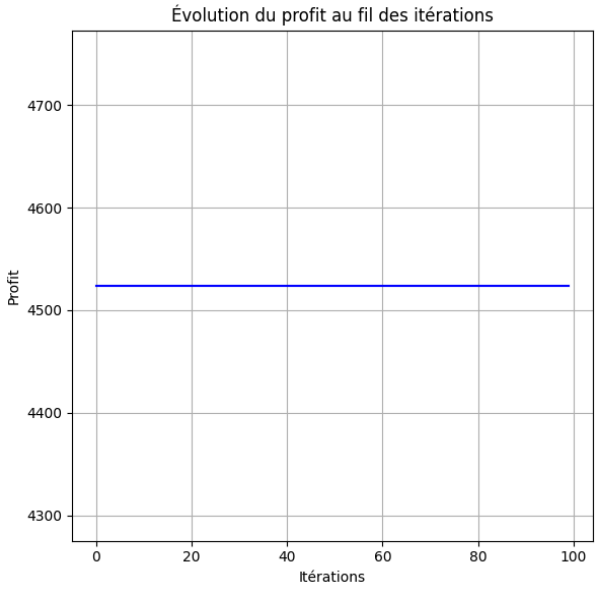


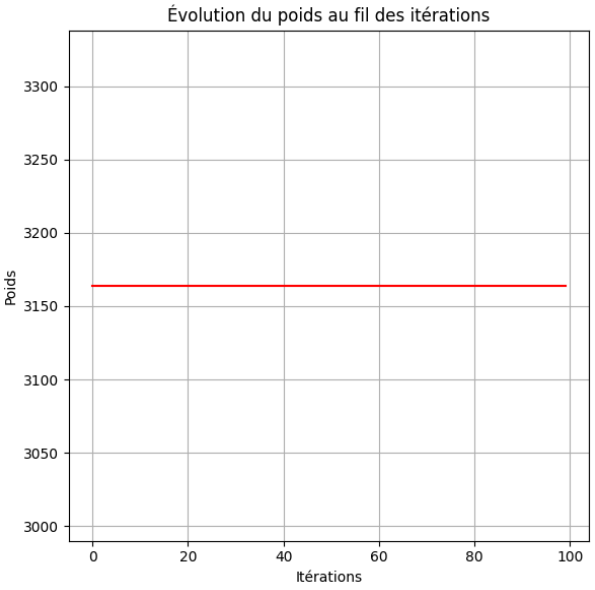




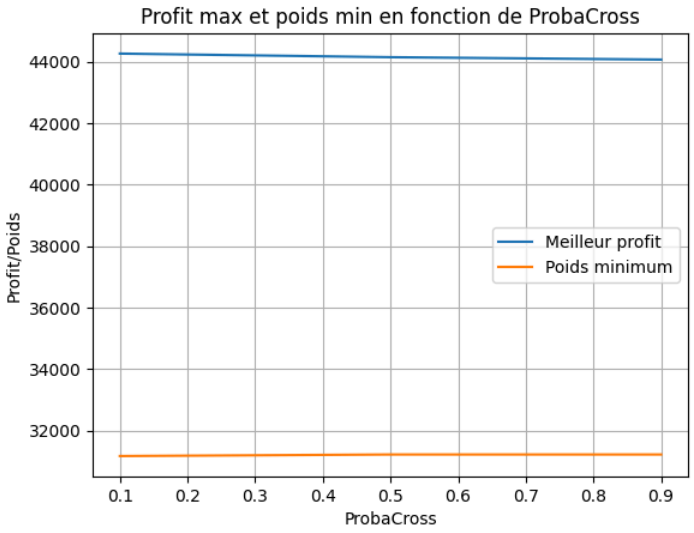
Pour le fichier pi13 avec n=1000, nous n'avons pas trouvé de solutions qui atteignent le même profit et poids que la solution donnée par le solver.

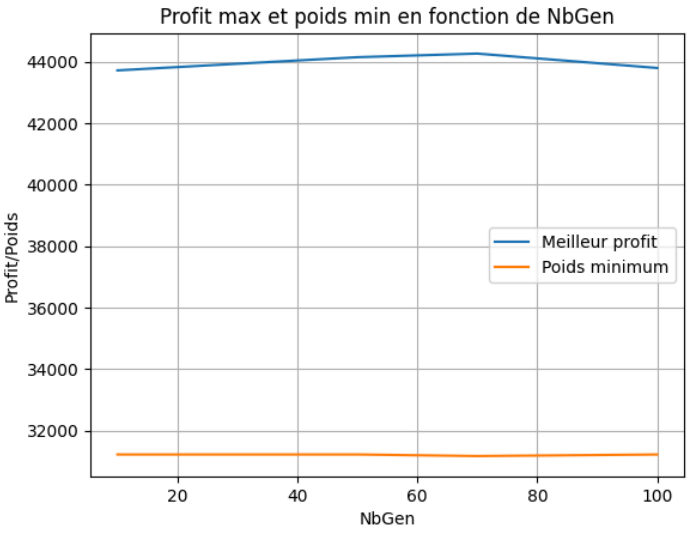
❖ n=10 000

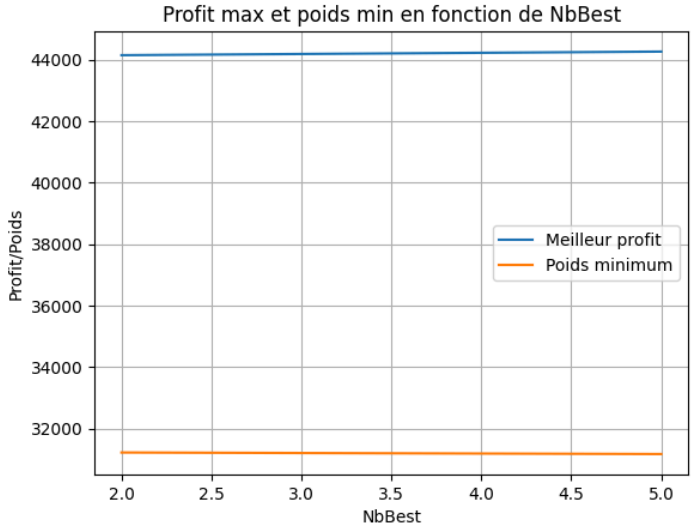


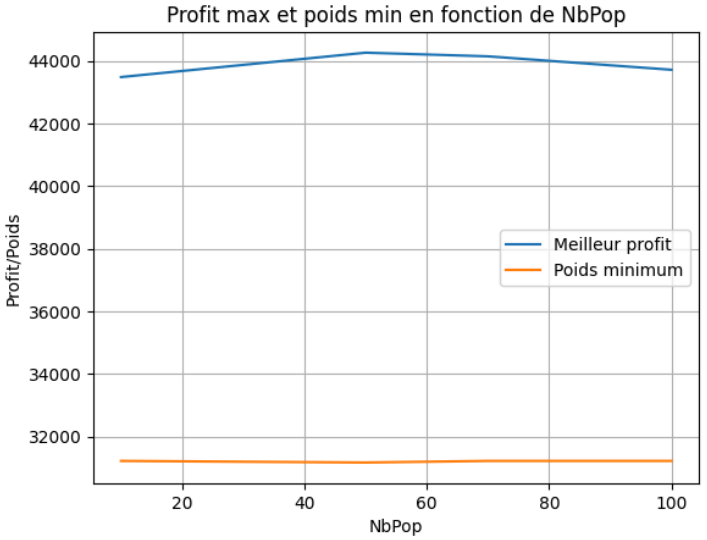


Graphiques:





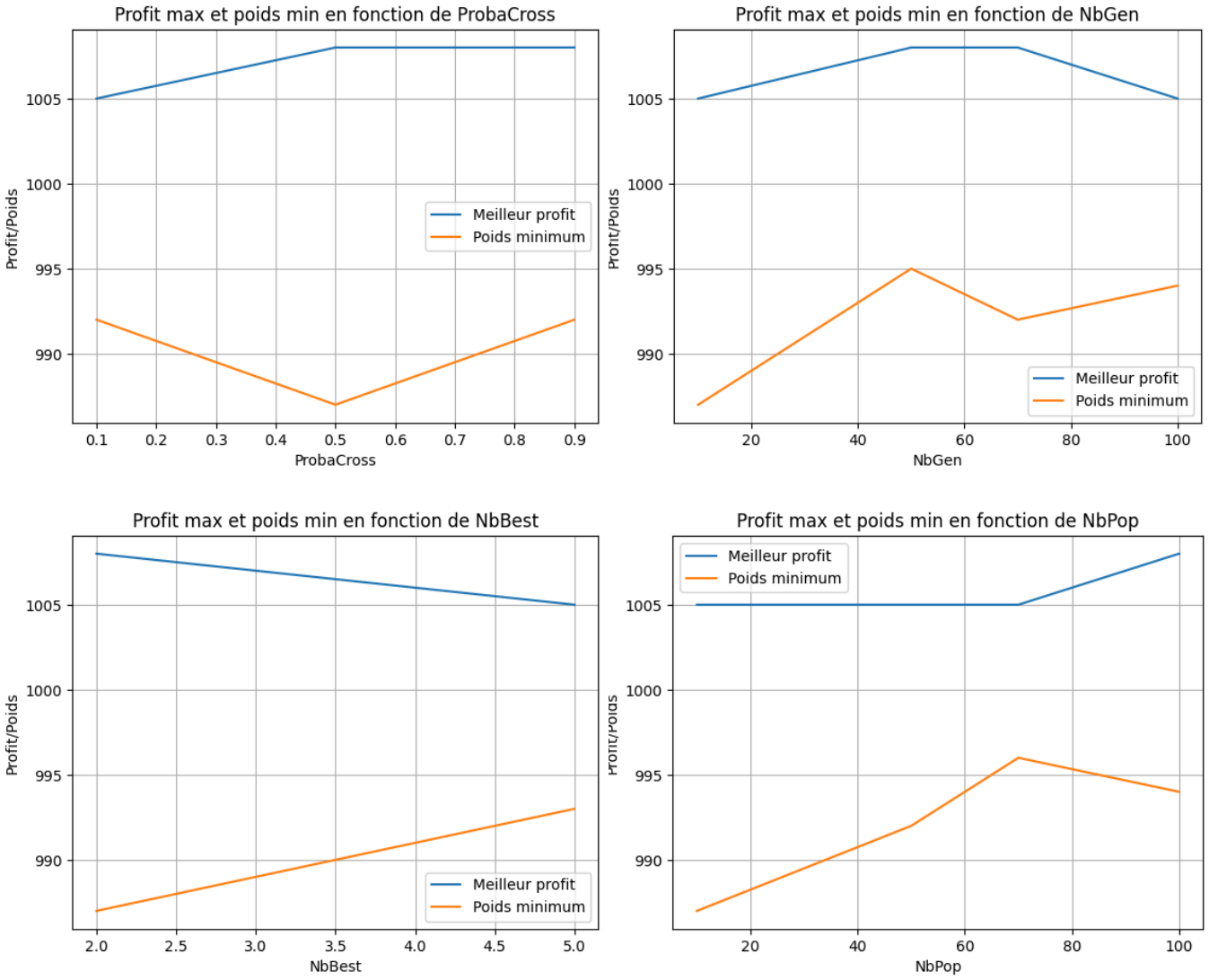




Nous n'avons pas trouvé de solutions qui atteignent le même profit et poids que la solution donnée par le solver.

★ RÉSULTATS PI-15:
- ❖ n=100

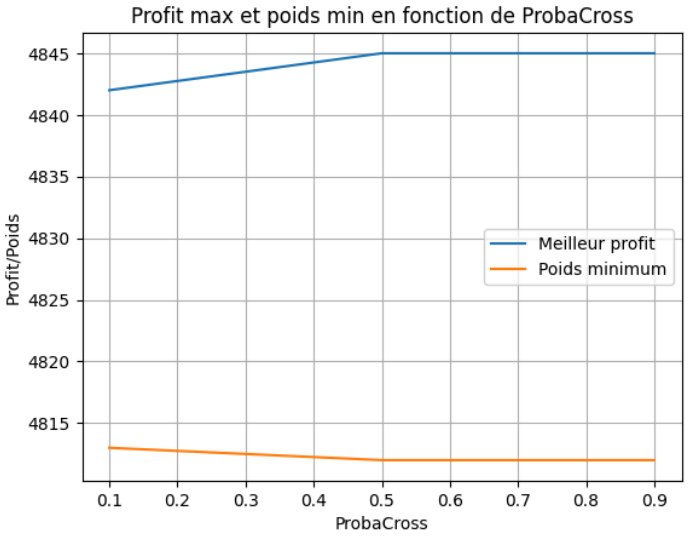
Graphiques:

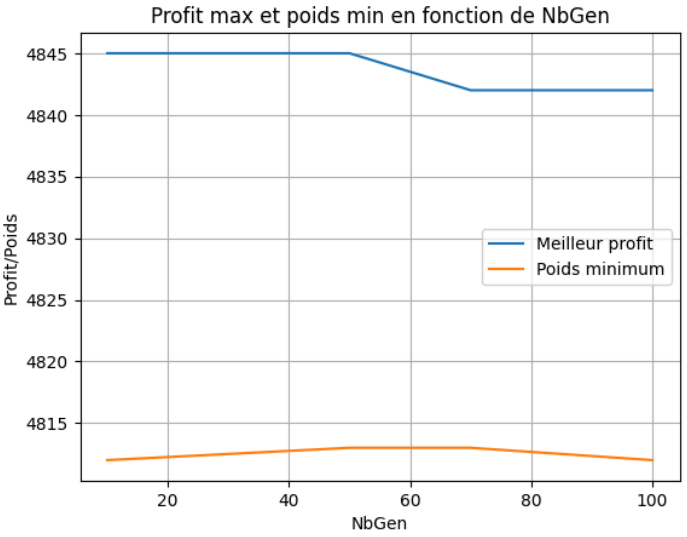


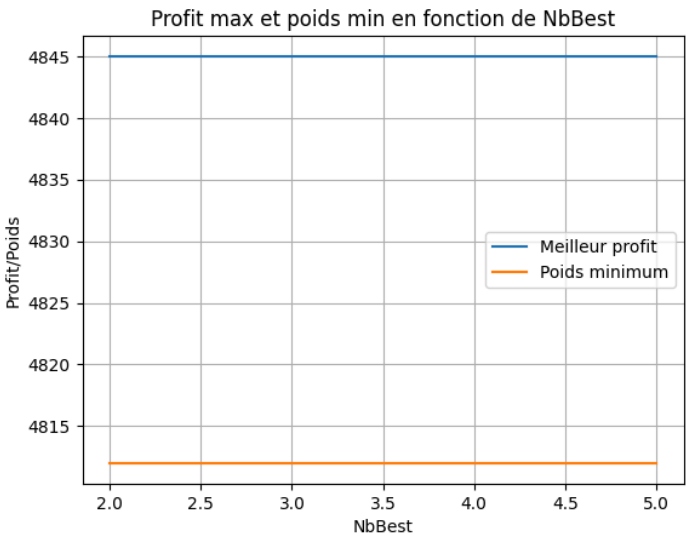
Pour le fichier pi15 avec n=100, nous n'avons pas trouvé de solutions qui atteignent le même profit et poids que la solution donnée par le solver.

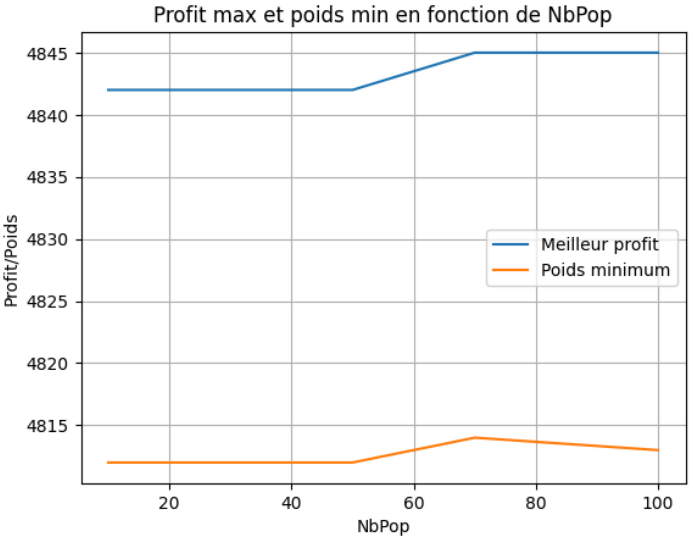
- ❖ n=1000

Graphiques:





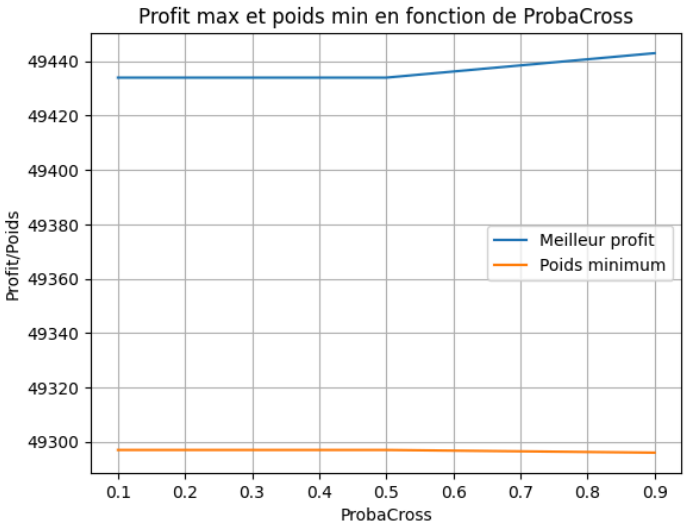


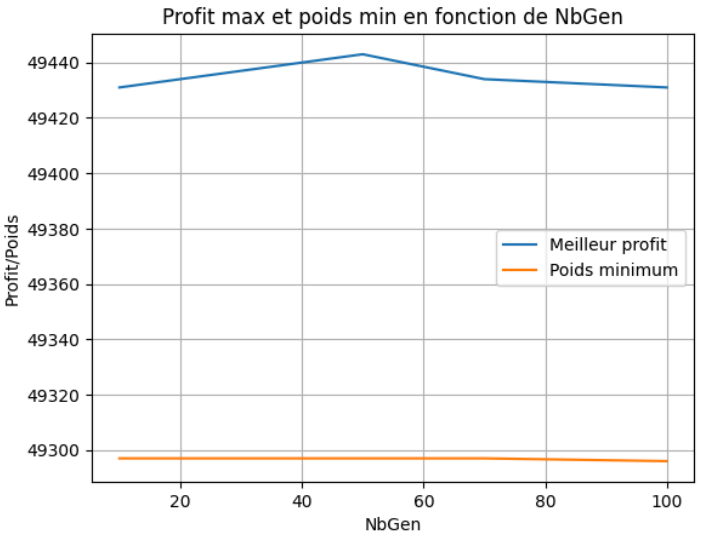


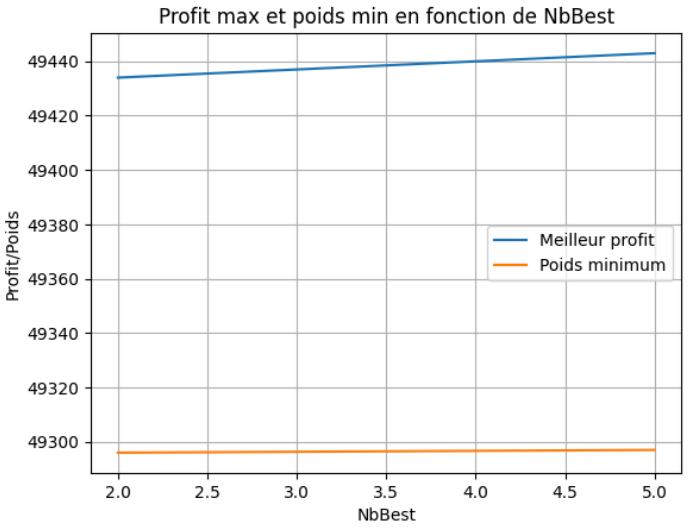
Pour le fichier pi15 avec n=1000, nous n'avons pas trouvé de solutions qui atteignent le même profit et poids que la solution donnée par le solver.

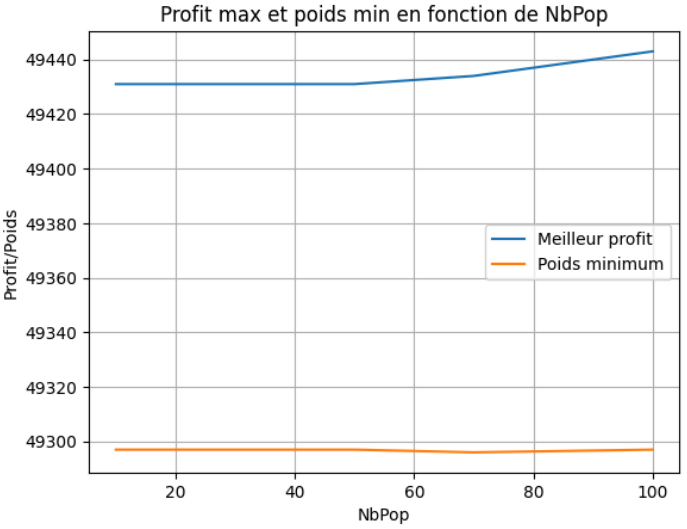
- n=10 000

Graphiques









Pour le fichier pi15 avec n=10 000, nous n'avons pas trouvé de solutions qui atteignent le même profit et poids que la solution donnée par le solver.

Interprétations des résultats obtenus par nos tests sur l'algo génétique

On observe que, tout comme le recuit simulé, l'algorithme génétique parvient systématiquement à produire une solution: aucun blocage n'a été constaté même après plusieurs itérations. Cela confirme la robustesse de l'approche en termes de convergence vers une solution réalisable.

Cependant, l'exploration de l'espace des solutions semble parfois limitée. En effet, dans plusieurs cas, le poids et profit de la solution finale obtenue reste identique à ceux de la solution initiale tirée aléatoirement ce qui signifie que l'algorithme n'a pas réussi à améliorer la solution initiale au fil des générations. Cela peut indiquer un manque de diversité

génétique dans la population ou un nombre d'itérations insuffisant pour permettre une véritable exploration de l'espace.

Parmi tous les tests effectués, seules deux configurations ont permis d'atteindre à la fois le profit et le poids exacts de la solution optimale fournie par le solver :

Le fichier pi12 avec n=100 et les paramètres suivants : NbPop = 100, NbGen = 100, probaCross = 0.9, NbBest = 2.

Le fichier pi13 avec n=100 et les paramètres : NbPop = 50, NbGen = 100, probaCross = 0.5, NbBest = 5.

Dans ces deux cas, le nombre de générations est de 100 ce qui laisse supposer que le nombre d'itérations (NbGen) joue un rôle central dans la capacité de l'algorithme à converger vers une solution optimale. En revanche, aucune tendance claire ne se dégage concernant les autres paramètres tels que la taille de la population, la probabilité de croisement ou le nombre de meilleurs individus conservés. Les résultats obtenus et les graphiques associés ne montrent aucune corrélation apparente entre ces paramètres et la qualité des solutions trouvées.

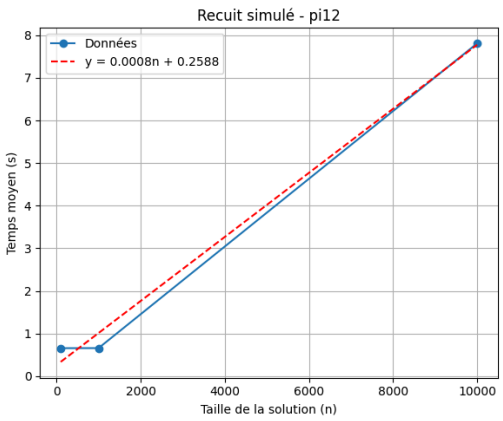
On observe également que seules les instances de taille n=100 pour tous les fichiers ont permis d'obtenir des profits maximaux dans les solutions finales. Cela rejoint l'hypothèse formulée dans l'analyse du recuit simulé: plus la taille des données est grande, plus l'espace de recherche est vaste et plus il devient difficile d'atteindre les solutions optimales. Le nombre de voisins potentiels augmente considérablement rendant l'exploration exhaustive plus difficile, surtout dans un nombre limité d'itérations.

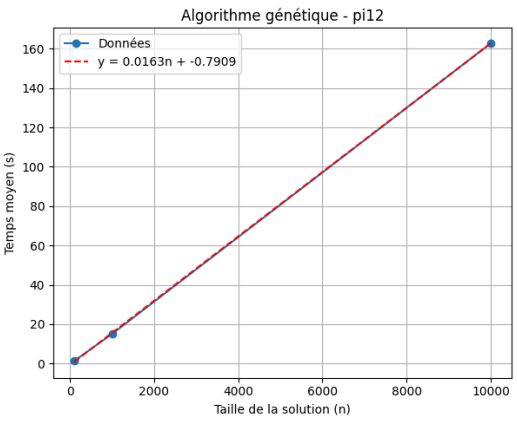
IV. Comparaison de temps d'exécution

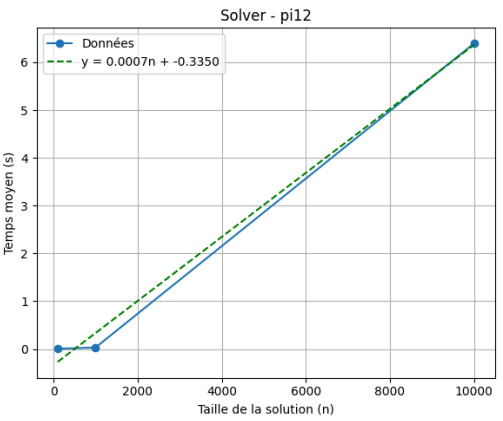
Dans cette partie, on compare les temps d'exécution des différentes métaheuristiques entre elles ainsi qu'avec celui du solver. De plus, on réalise une régression linéaire pour déterminer si le temps d'exécution dépend de la taille de la solution. Étant donné que les temps d'exécution ne sont pas identiques pour chaque solution, on a calculé la moyenne des temps pour chaque taille de solution (n=100, 1000 puis 10k).

Voici les résultats obtenus :

★ Pi-12



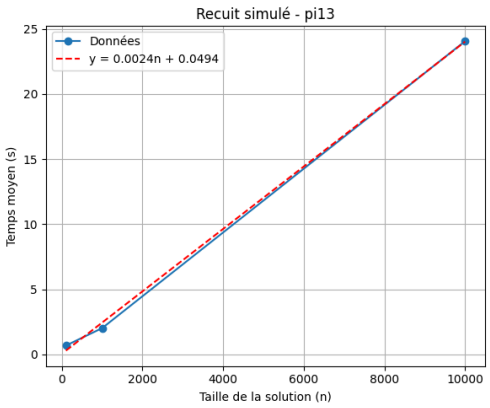


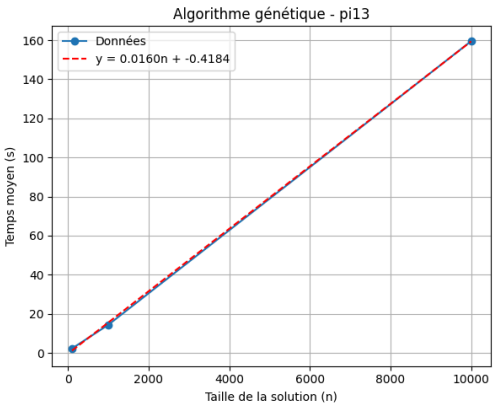


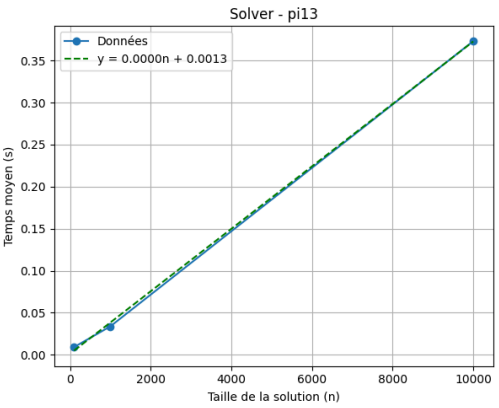
Méthode utilisée	Taille des solutions (n) - Temps (en secondes)		
	100	**1000**	**10 000**
Recuit simulé	0.655	0.656	7.808
Algo génétique	1.329	15.018	162.695
Solver	0.007	0.032	6.394

Tableau 3: Temps d'exécution pour chaque méthode utilise (en secondes) pour les données du fichier pi12

★ Pi-13



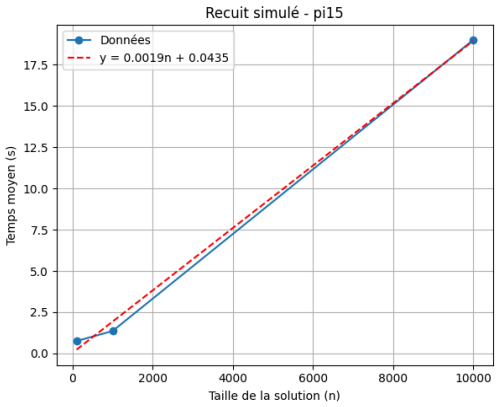


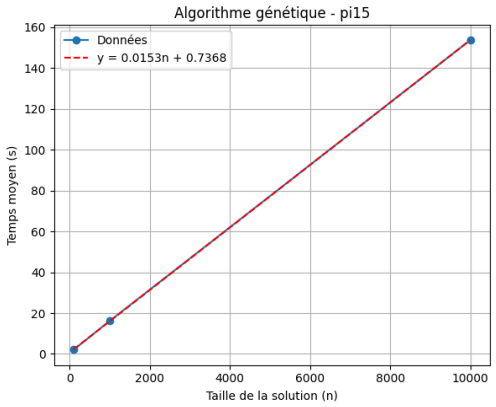


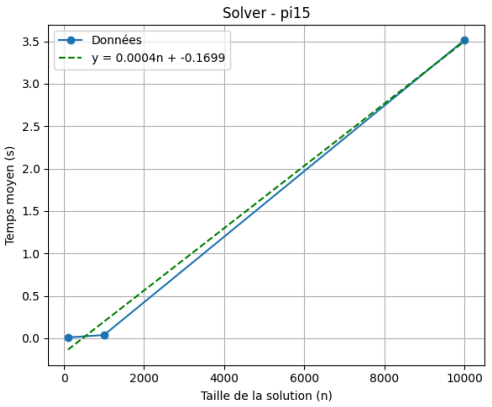
Méthode utilisée	Taille des solutions (n) - Temps (en secondes)		
	100	**1000**	**10 000**
Recuit simulé	0.677	2.020	24.065
Algo génétique	2.268	14.365	159.502
Solver	0.009	0.033	0.372

Tableau 4: Temps d'exécution pour chaque méthode utilise (en secondes) pour les données du fichier pi13

★ Pi-15







Méthode utilisée	**Taille des solutions (n) - Temps (en secondes)**		
	100	**1000**	**10 000**
Recuit simulé	0.752	1.361	18.995
Algo génétique	2.103	16.212	153.677
Solver	0.010	0.038	3.514

Tableau 5: Temps d'exécution pour chaque méthode utilise (en secondes) pour les données du fichier pi15

On remarque que le solver fournit une solution quasi instantanée pour des tailles de problème relativement petites comme n = 100 ou même n = 1000. Cependant, dès que n devient suffisamment grand, le temps d'exécution augmente fortement.

Par exemple, lorsque l'on passe de n = 100 à n = 1000, soit un facteur 10, le temps d'exécution augmente d'environ 300 %. En revanche, lorsqu'on passe de n = 1000 à n = 10 000, toujours un facteur 10, le temps augmente de près de 1500 %, ce qui montre que la croissance du temps d'exécution n'est pas linéaire. Cela suggère une complexité algorithmique polynomiale ou exponentielle ce qui est cohérent avec le fait que le problème du sac à dos est NP-difficile.

On constate également que les temps d'exécution de l'algorithme génétique sont nettement plus élevés que ceux du recuit simulé pour une même taille de solution. Pourtant, les deux méthodes aboutissent souvent à des solutions équivalentes lorsqu'elles sont configurées de manière similaire. Le recuit simulé parvient simplement à ces résultats plus rapidement, ce qui le rend plus efficace en pratique dans ce contexte.

Conclusion

Dans ce projet, nous avons implémenté deux métaheuristiques pour résoudre le problème bien connu du sac à dos dont l'objectif est de maximiser le profit tout en respectant une contrainte de poids à ne pas dépasser. En testant ces deux approches tout en faisant varier leurs paramètres, nous avons observé que les résultats dépendent fortement de la solution initiale générée de manière aléatoire. Ainsi, pour avoir de meilleures chances d'atteindre le profit maximal, il est judicieux de réaliser plusieurs essais avec différentes solutions de départ. Cependant, cette stratégie peut prendre beaucoup de temps.

Les deux métaheuristiques parviennent globalement à produire des solutions de bonne qualité qui atteignent le profit maximal dans de nombreux cas tout en respectant la contrainte de poids. Toutefois, elles atteignent rarement le poids minimal possible, celui donné par la solution optimale fournie par le solver. On peut en déduire que, du point de vue du profit (notre objectif principal), les deux méthodes sont efficaces et comparables en termes de qualité de solution. En revanche, sur le plan du temps d'exécution, le recuit simulé se démarque clairement : il atteint ces bons résultats beaucoup plus rapidement que l'algorithme génétique.

Il convient également de souligner que l'algorithme génétique dispose de davantage de paramètres que le recuit simulé, ce qui peut potentiellement le rendre plus précis mais aussi plus complexe à régler. De plus, le recuit simulé n'a pas besoin qu'on lui spécifie un nombre d'itérations : il s'arrête de lui-même lorsqu'il atteint une solution optimale ou lorsqu'il ne trouve plus d'amélioration significative. En revanche, l'algorithme génétique nécessite la définition préalable du nombre de générations à exécuter.

Enfin, il est important de noter une différence conceptuelle majeure entre les deux : le recuit simulé est un algorithme sans mémoire ce qui le rend plus simple mais potentiellement moins efficace pour éviter les cycles ou les répétitions. À l'inverse, l'algorithme génétique intègre une mémoire implicite à travers la population qu'il maintient à chaque génération. Les caractéristiques des meilleures solutions sont ainsi transmises d'une génération à l'autre via les opérateurs de sélection, croisement et élitisme. Cela en fait un algorithme avec mémoire, à l'image de la méthode tabou et cela peut jouer un rôle dans la diversité et l'exploration de l'espace de recherche.

Bibliographie

[1] Benford, S. L. (2021). *Solving the binary knapsack problem using tabular and deep reinforcement learning algorithms* (Thèse de master, Northeastern University, Department of Mechanical and Industrial Engineering).

[2] Optimisations_discrètes: Repository GitHub contenant le code source, les tableaux de résultats et l'ensemble des fichiers utilisés dans notre projet