

Benchmark des méthodes de résolution du problème du sac à dos (Knapsack 0/1)

Présenté par:

- Safae Ahrara
- Chaimae El Amraoui



I. Formalisation du Problème

A. Présentation du Problème du Sac à Dos

- Le problème du sac à dos, ou Knapsack Problem, est un défi d'optimisation Classique.
- Il s'agit de sélectionner un sous-ensemble d'objets, chacun ayant un poids et une valeur, pour les placer dans un sac à dos de capacité limitée.
- L'objectif est de maximiser la valeur totale des objets transportés sans dépasser la capacité maximale du sac.
- C'est un problème NP-difficile, ce qui signifie qu'il n'existe pas de solution polynomiale connue pour le résoudre.



B. Formulation CSP (Constraint Satisfaction Problem) du Sac à Dos

CSP du Knapsack = (X, D, C, F)

- **Variables X** : $x_i \in \{0,1\}$ pour chaque objet

- **Domaines D** : $x_i \in \{0,1\}$

- **Contrainte C** :

$$\sum_{i=1}^n w_i x_i \leq W$$

- **Objectif F** :

$$\max \sum_{i=1}^n v_i x_i$$

C. Solveurs Implémentés et Métriques de Comparaison

Pour cette étude, plusieurs solveurs ont été implémentés en Python, permettant une comparaison rigoureuse des performances.

Comparaison des Méthodes

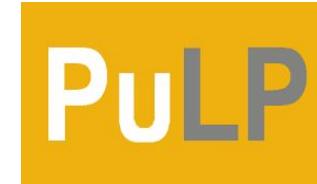
- Méthodes complètes (exactes)
- Méthodes incomplètes (heuristiques)



ORTOOLS

Métriques Mesurées

- Valeur obtenue
- Temps de calcul
- Écart (Gap) par rapport à l'optimal



PuLP

II. Benchmark d'Instances

Instances de Test

Un jeu de 30 instances a été utilisé pour le benchmark, couvrant différentes tailles de problèmes afin d'évaluer la scalabilité des méthodes.

- **Nombre d'instances :** 30
- **Tailles des instances :** 50, 100, 1000 objets
- **Format d'entrée :** Fichiers .kp (format standardisé)

Chaque fichier .kp est structuré de manière simple pour faciliter le parsing :

100
500
(20, 100)
(30, 150)
(10, 60)
(50, 200)

...
Ligne 1 : Nombre d'objets

Ligne 2 : Capacité du sac à dos

Lignes suivantes : Paires (poids, profit) pour chaque objet

II. Benchmark d'Instances

Les types d'instances

Le projet utilise la bibliothèque **kplib** qui propose plusieurs types d'instances :

1. **Uncorrelated** : Poids et profits indépendants (2 instances)
2. **Weakly Correlated** : Corrélation faible entre poids et profits (2 instances)
3. **Strongly Correlated** : Profits proportionnels aux poids (1 instance)
4. **Inverse Strongly Correlated** : Corrélation inverse (1 instance)
5. **Almost Strongly Correlated** : Proche de la corrélation forte (1 instance)
6. **Subset Sum** : Cas particulier où $p_i = w_i$ (1 instance)
7. **Uncorrelated with Similar Weights** : Poids similaires (1 instance)
8. **Spanner** : Instances avec structure particulière (1 instance)

III. Méthodes de Résolution

A. Approches Complètes (Vue d'ensemble)

Les approches complètes garantissent de trouver la solution optimale, mais leur complexité augmente exponentiellement avec la taille du problème.

Programmation Dynamique

Résolution par sous-problèmes.

MIP (PuLP)

Programmation en Nombres Entiers Mixtes.

Branch & Bound

Exploration arborescente avec élagage.

MIP (OR-Tools)

Suite d'outils d'optimisation de Google.

MIP (Programmation en Nombres Entiers Mixtes)

La Programmation en Nombres Entiers Mixtes, ou MIP, est une méthode qui modélise le problème avec des variables binaires et des contraintes mathématiques, puis utilise un solveur externe pour trouver la solution optimale.

- Garantit l'optimalité de la solution trouvée.
- Adapté aux instances petites et moyennes.
- Solutions fiables et interprétables.
- Temps de calcul très élevé pour les grandes instances.
- Peut devenir prohibitif en mémoire et en ressources CPU.
- Peut produire des timeouts sur des instances avec $n = 1000$.

Exemple

$$4x_A + 3x_B + 2x_C \leq 5$$

$$\max(10x_A + 7x_B + 6x_C)$$

capacité de sac : 5

Objet	Poids	Valeur
A	4	10
B	3	7
C	2	6

Pour chaque objet, tu définis une variable :

- $X_A=1$ si tu prends A, sinon 0
- $X_B=1$ si tu prends B, sinon 0
- $X_C=1$ si tu prends C, sinon 0

x_A	x_B	x_C	Poids total	Valeur
1	0	0	4	10
0	1	1	5	13 meilleur
1	1	0	7 invalide	
0	0	1	2	6

Branch and Bound

Le Branch and Bound est une technique d'exploration arborescente qui construit un arbre de décision pour trouver la solution optimale.

- Garantit l'**optimalité** de la solution finale.
- Réduit considérablement l'espace de recherche grâce au **pruning** (**élagage**).
- Plus efficace qu'une recherche exhaustive brute.
- Peut résoudre des instances de taille moyenne avec de bonnes bornes.
- Le temps de calcul peut devenir **très élevé** pour les grandes instances.
- Peut explorer un très grand nombre de nœuds dans le pire des cas.
- Performances très dépendantes de la qualité des bornes et de l'ordre des objets.
- Peut atteindre des **timeouts** pour $n = 1000$.
- Complexité exponentielle dans le cas général.

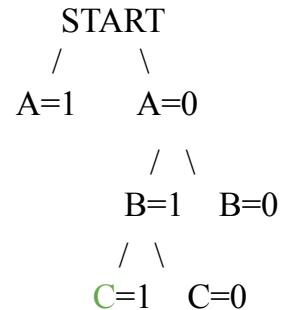
Exemple

$$4x_A + 3x_B + 2x_C \leq 5$$

$$\max(10x_A + 7x_B + 6x_C)$$

capacité de sac : 5

Objet	Poids	Valeur
A	4	10
B	3	7
C	2	6



Programmation Dynamique

La Programmation Dynamique résout le problème en le décomposant en sous-problèmes plus petits, stockant les solutions intermédiaires dans une table.

- Garantit **la solution optimale**.
- Méthode déterministe et stable (donne toujours le même résultat).
- Facile à implémenter et à expliquer pédagogiquement.
- Bien adaptée aux problèmes structurés comme le sac à dos 0/1.
- **Explosion mémoire** quand la capacité est grande (table DP très large).
- Temps de calcul proportionnel à $n \times$ **capacité** (peut être énorme).
- Impraticable pour des capacités de l'ordre du million.
- Moins adapté aux très grandes instances que MIP avec bons solveurs.

Exemple

$$4x_A + 3x_B + 2x_C \leq 5$$

$$\max(10x_A + 7x_B + 6x_C)$$

capacité de sac : 5

Objet	Poids	Valeur
A	4	10
B	3	7
C	2	6

Résultat :

Objets \ Capacité

0 (aucun objet)

C (poids 2, val 6)

B + C

A + B + C

0 1 2 3 4 5

0 0 0 0 0 0

0 0 6 6 6 6

0 0 6 7 7 13

0 0 6 7 10 13

B. Approches Incomplètes (Vue d'ensemble)

Les approches incomplètes, ou heuristiques, ne garantissent pas l'optimalité mais fournissent des solutions de très bonne qualité en un temps de calcul considérablement réduit, ce qui les rend pratiques pour les grandes instances.



Greedy Simple

Approche gloutonne basée sur un ratio.



Reclut Simulé

Méthode d'optimisation inspirée de la physique.



Algorithme Génétique

Optimisation inspirée de l'évolution biologique.



Recherche Tabou

Exploration de l'espace de recherche avec mémoire.

Glouton Simple, Aléatoire k3, Probabiliste $\alpha=0.9$

Le Greedy Simple agit comme un voyageur pressé : il choisit toujours l'objet le plus rentable en premier, ce qui le rend ultra rapide, mais parfois myope face à la meilleure solution globale.

Méthode	Image mentale	Comment elle choisit	Résultat
Greedy Simple	Robot pressé	Toujours le meilleur	Fixe, prévisible
Greedy Random k=3	Aventurier	Hasard parmi top 3	Très variable
Greedy Probabiliste	Stratège	Hasard MAIS biaisé vers le meilleur	Bon compromis

Recuit Simulé : Refroidir pour Optimiser

Le Recuit Simulé part d'une solution initiale et l'améliore progressivement en acceptant parfois de mauvaises décisions pour éviter de rester bloqué dans un optimum local.

Avantages

- Bonne qualité de solution (Gap moyen : 0.24%)
- Taux optimal très élevé (93.3%)
- Capacité à s'échapper des optima locaux

Inconvénients

- Temps de calcul modéré (129 ms)
- Sensible aux paramètres de "température" et de "refroidissement"

Recherche Tabou : Apprendre de l'Expérience

La Recherche Tabou explore systématiquement des solutions voisines tout en conservant une mémoire des choix récents afin d'éviter les répétitions et guider intelligemment la recherche vers de nouvelles zones prometteuses



Mémoire

Évite les boucles grâce à la liste taboue.



Exploration

Permet de découvrir de nouvelles solutions.



Performance

Gap moyen : 0.24%, Taux optimal : 93.3%.

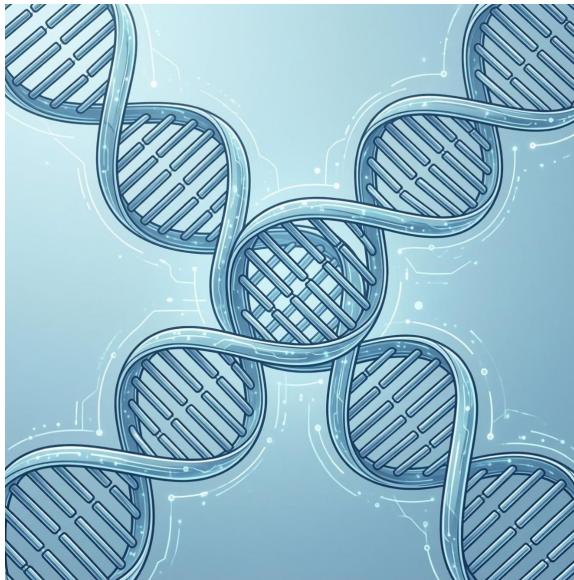


Rapidité

Très efficace avec un temps moyen de 79 ms.

Algorithme Génétique : La Sélection Naturelle en Action

L'Algorithme Génétique est une méthode inspirée de l'évolution naturelle : il travaille sur plusieurs solutions à la fois (une population), sélectionne les meilleures, les combine entre elles (crossover) et introduit des changements aléatoires (mutation) pour améliorer progressivement les solutions au problème du sac à dos.



1

Puissance

Excellent pour explorer de vastes espaces de recherche.

2

Adaptabilité

Très flexible pour différents types de problèmes.

Résultats et Analyse Détailée

Méthode	Type	Valeur moyenne	Temps moyen (ms)	Taux optimal (%)	Gap moyen (%)
Complete_DP	Complète	18850	1616	100.0	0.00
Complete_MIP_ORTools	Complète	117877	20270	93.3	0.00
Complete_MIP_PuLP	Complète	25268	176	100.0	0.00
Incomplete_GeneticAlgorithm	Incomplète	146534	3462	10.0	7.98
Incomplete_Greedy_Probabilistic	Incomplète	106321	239	6.7	9.51
Incomplete_Greedy_Random_k3	Incomplète	117775	226	86.7	0.41
Incomplete_Greedy_Simple	Incomplète	117812	2	93.3	0.25
Incomplete_SimulatedAnnealing	Incomplète	117854	2547	100.0	0.06
Incomplete_TabuSearch	Incomplète	117827	103503	93.3	0.19

Les données présentées dans ce tableau sont les moyennes sur 30 instances. Pour des raisons de concision, l'écart-type est omis du tableau, mais il a été pris en compte dans l'analyse détaillée de chaque méthode.

Analyse par Type de Méthode

Méthodes Complètes

MEILLEURE : PULP

PuLP: Optimalité garantie (100%), très rapide (176 ms), très stable. Idéal pour $n \leq 100$.

OR-Tools: Performant sur instances **moyennes/grandes** ($100 < n < 500$), avec quelques timeouts.

DP: Optimal pour les **petites instances** (capacité $\leq 10\,000$) et temps modéré (1.6s).

Méthodes Incomplètes (Heuristiques)

MEILLEURE : RECUIT SIMULÉ

Requit Simulé

Qualité : 0.06% de gap, 100% optimal.

Temps : 2.5s. La meilleure qualité.

Greedy Simple

Qualité : 0.25% de gap, 93.3% optimal.

Temps : 2 ms. Le plus rapide.

À Éviter

L'Algorithme Génétique et le Greedy Probabiliste ont des gaps élevés et sont peu optimaux.

Robustesse et Comportement par Taille d'Instance

Instances Faciles (n=50)

- Toutes les méthodes complètes atteignent l'optimum.
- Le Greedy Simple est quasi-optimal (99%).
- Les temps d'exécution sont inférieurs à 1 seconde pour toutes les approches.

Instances Moyennes (n=100)

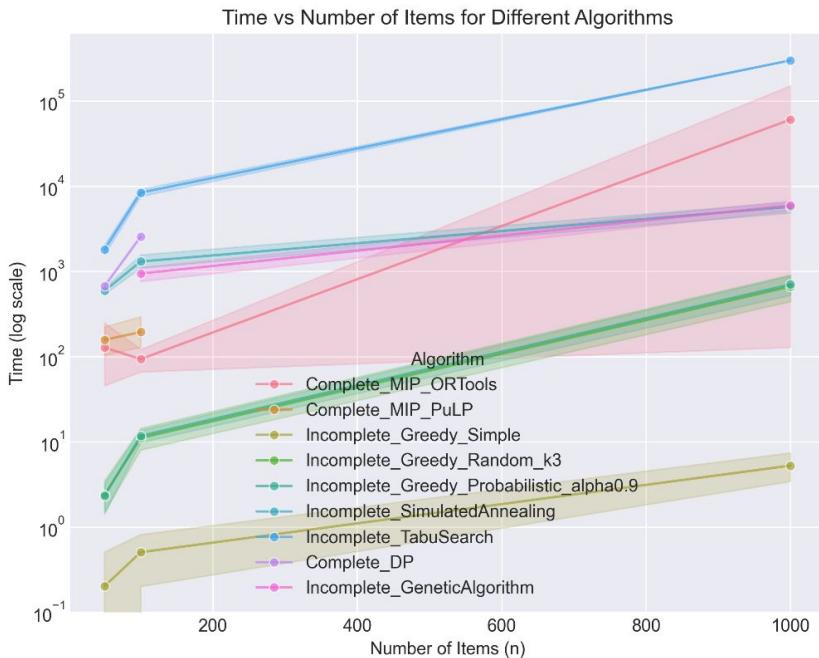
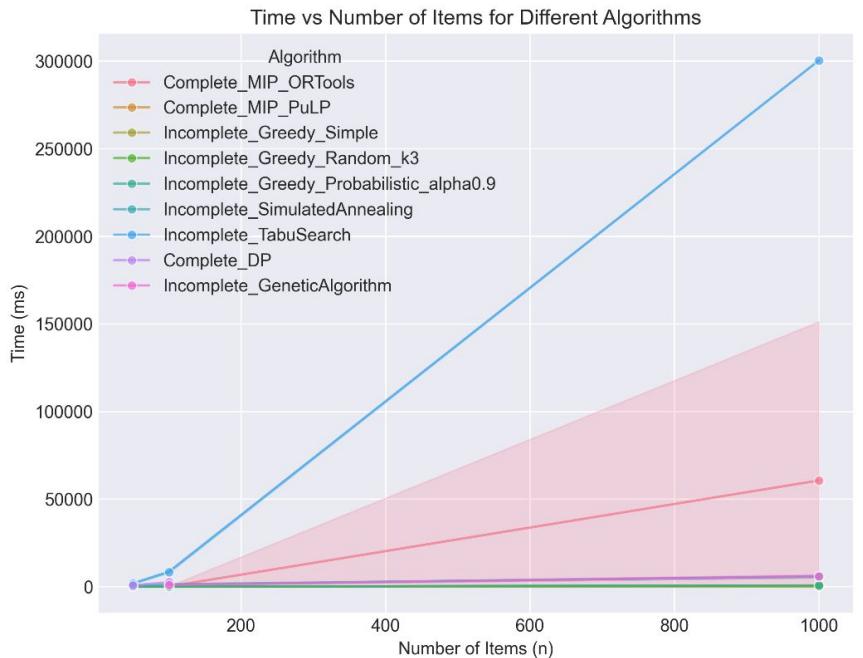
- PuLP et la Programmation Dynamique restent optimaux.
- OR-Tools rencontre un timeout sur 10 instances.
- Le Recul Simulé (100% optimal) et Greedy Simple (90% optimal) maintiennent de bonnes performances.

Instances Difficiles (n=1000)

- Seul OR-Tools parvient à terminer (avec 20% de timeouts).
- Le Recul Simulé reste très performant (gap < 0.1%), tout comme le Greedy Simple (gap < 0.5%).

Visualisation des Compromis

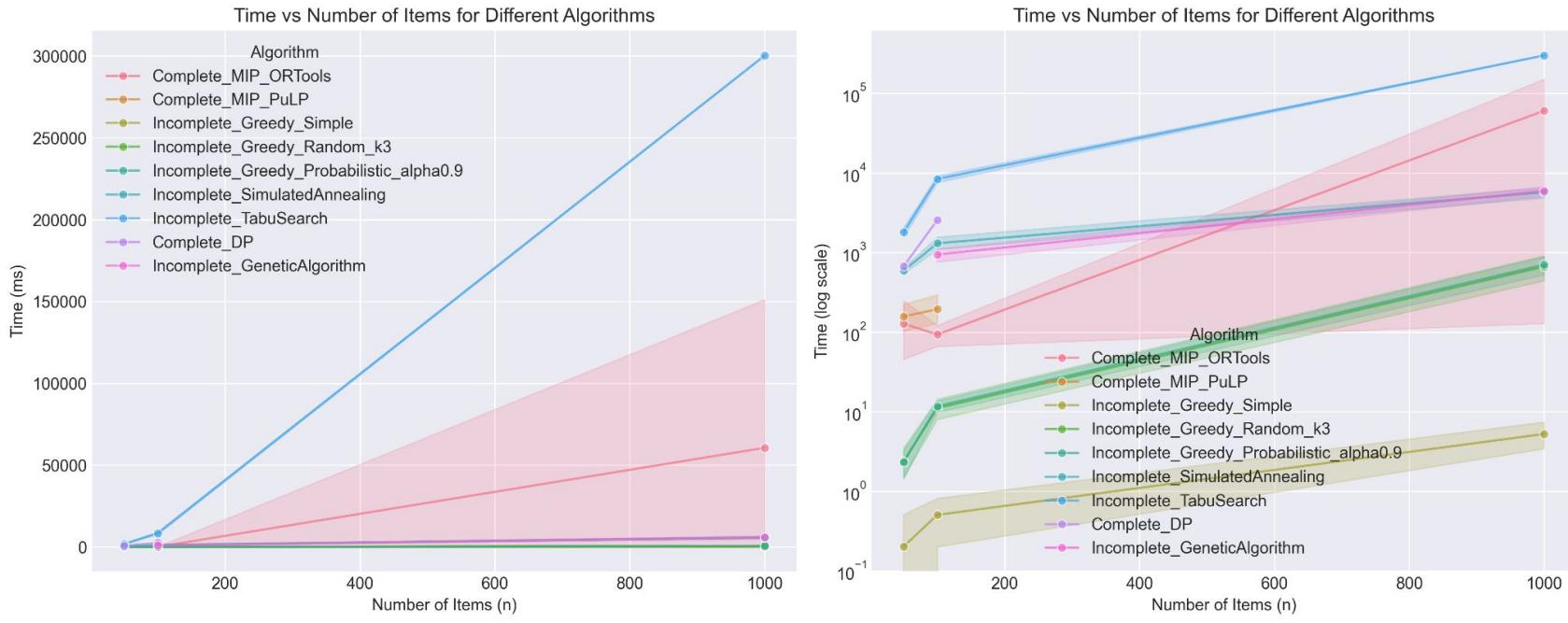
- Certaines courbes restent **quasiment collées à zéro**
- D'autres explosent très rapidement dès que n augmente



Temps d'exécution en fonction du nombre d'objets

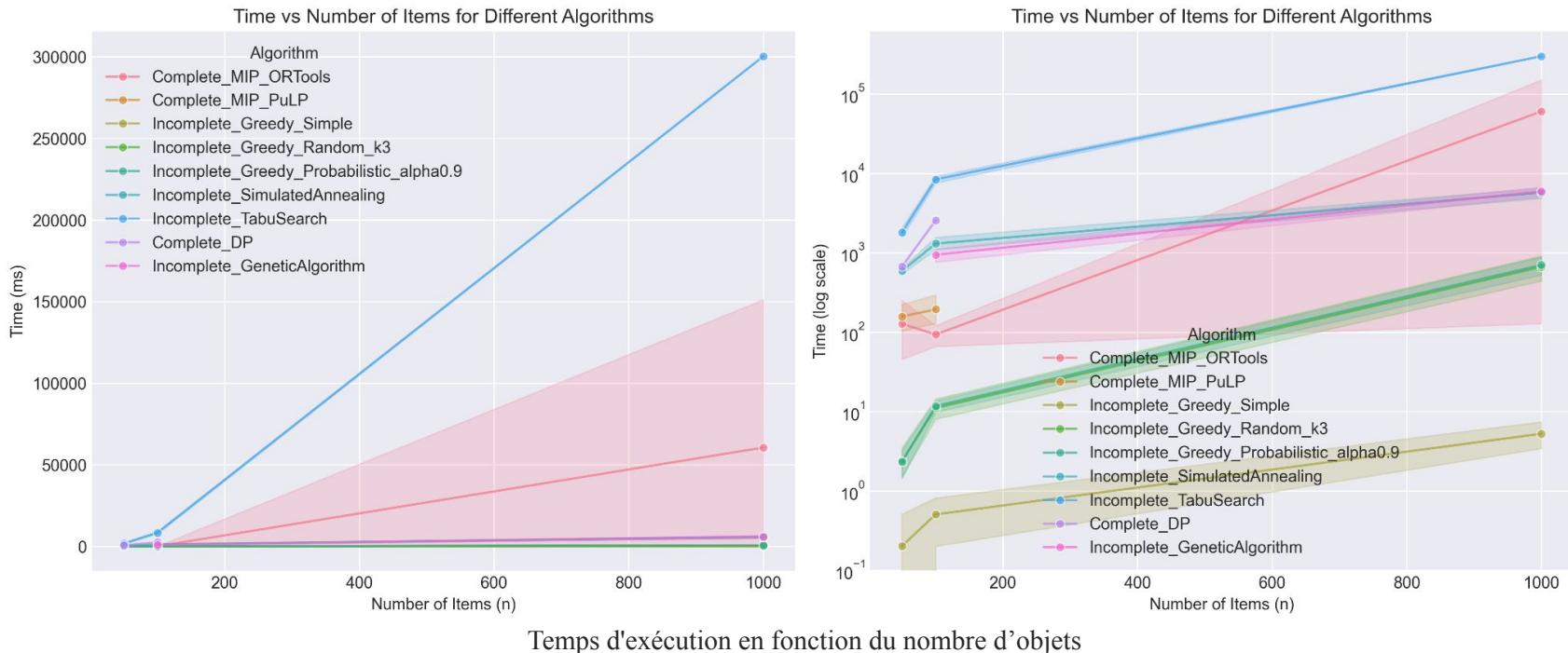
Visualisation des Compromis

- PuLP reste relativement rapide et stable jusqu'à $n = 100$
- OR-Tools a une croissance beaucoup plus rapide et très variable
- Programmation dynamique augmente de manière régulière mais ne passe pas à l'échelle



Visualisation des Compromis

- Les méthodes **Greedy** restent extrêmement rapides, même pour $n = 1000$
- Le **recuit simulé** augmente progressivement mais reste sous quelques secondes
- La recherche **Tabu** est très coûteuse et devient impraticable pour les grandes tailles



Conclusions et Recommandations

Tableau de Décision Final

Ce tableau récapitule les recommandations basées sur des critères clés pour le choix de la méthode de résolution.

Critère	Méthode Recommandée	Justification
Optimalité garantie	PuLP	100% optimal, rapide
Grandes instances	OR-Tools	Seul à terminer sur n=1000
Vitesse maximale	Greedy Simple	2 ms en moyenne
Meilleur gap	Reclut Simulé	0.06% en moyenne
Stabilité	PuLP	Variance minimale des temps
Compromis Vitesse/Qualité	Greedy Simple	Très rapide + quasi-optimal