Projet de Métaheuristiques

Maxime APARICIO et Bruno GLIBERT

Novembre 2025

Contents

1 Introduction

1.1 Description du problème

Ce projet porte sur un problème classique en optimisation combinatoire : le problème de localisation discrète. La question est de décider comment répartir N clients entre M usines, sachant que sont donnés :

- les prix d'ouverture de chacune des M usines, $(P_j)_{1\leq j\leq M}$,
- ainsi que les prix d'affectation de chaque client i à chaque usine j, $(A_{ij})_{1\leq i\leq N,\ 1\leq j\leq M}$.

Deux versions du problème seront abordées ici :

1. une version "simple", où chaque usine a une capacité d'accueil illimitée,
2. et une version "difficile", où sont fournis en plus des prix ci-dessus :
   - la capacité d'accueil (finie) de chaque usine,
   - et la demande (ou besoin de production) de chaque client.

Le problème "simple" est traité entre les Sections 2 et 5, et le problème "difficile" est abordé à la Section 6.

1.2 Définitions et notations

On commence par introduire quelques définitions et notations utilisées tout au long du document.

- N désignera toujours le nombre de clients du problème,
- et M le nombre d'usines.
- Les prix d'ouverture des usines seront toujours regroupés au sein d'un vecteur noté $P \in \mathbb{R}^M$.
- De même, les prix d'affectations de tout client i à toute usine j seront toujours regroupés dans la matrice $A \in \mathcal{M}_{N,M}(\mathbb{R})$.
- Dans les algorithmes comme dans le texte, la variable i sera toujours utilisée pour désigner un client,
- et de la même façon, j désignera systématiquement une usine.
- Par "instance", on entend la donnée des prix P (prix d'ouverture) et A (prix d'affectation). Notons que dans la Section 6, le terme d'instance englobera en plus les capacités des usines et demandes des clients.

- Le terme "solution" sera employé pour désigner toute répartition possible des N clients au sein des M usines (que cette répartition minimise le coût total ou non). Une discussion sur la manière de porter l'information de cette répartition (question de la modélisation de la solution) sera conduite à la section 2.1.

- On parlera d'une "évaluation" pour calculer le coût total nécessaire à la mise en place d'une solution (*i.e.* la somme des prix d'ouvertures des usines utilisées et des prix d'affectations des clients à leur usine respective).

- Le coût d'affectation d'un client à une usine peut aussi bien s'interpréter comme un prix financier que comme une distance. Ainsi, on parlera volontiers de "l'usine la plus proche" d'un client i pour désigner l'usine à laquelle il revient le moins cher d'affecter i (*i.e.* : $\arg\min_{1 \leq j \leq M} A_{ij}$).

2 Heuristique initiale

2.1 Modélisation des solutions

Le problème de localisation discrète consistant à répartir N clients entre M usines, la représentation la plus naturelle pour une solution consiste à dresser une matrice booléenne (ou, de manière équivalente, binaire) d'affectations : N lignes pour les clients, M colonnes pour les usines (ou inversement), et marquer l'élément en position (i, j) comme Vrai (ou 1) si et seulement si le client i est placé dans l'usine j.

Par exemple, pour une instance à 3 clients et 2 usines, la solution consistant à placer

- les clients 1 et 2 à la 1^{ere} usine,
- et le client 3 à la 2^{eme},

peut être modélisée par la matrice :

$$M_{solution} = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Modéliser nos solutions grâce à leur matrice d'affectation a l'avantage d'établir une bijection : à une solution est associée une seule matrice, et réciproquement. La contre-partie de cette exhaustivité est la lourdeur du modèle : pour représenter une solution, il est nécessaire d'allouer $N \times M$ éléments.

C'est pourquoi on préférera pour nos solutions une autre modélisation, plus concise : porter pour seule information la liste des usines ouvertes. On ne manipule alors qu'un vecteur binaire de taille M, dont le j^{eme} élément est 1 si et seulement si l'usine j est ouverte dans la solution. Reprenons l'exemple ci-dessus aux 3 clients et 2 usines. Puisque, dans la solution décrite, les deux usines sont ouvertes, la solution est modélisée par le vecteur :

$$V_{solution} = \begin{pmatrix} 1 & 1 \end{pmatrix}$$

On remarque immédiatement qu'on perd la bijection que garantissait l'utilisation de matrices d'affectations : plusieurs solutions sont maintenant représentées par le même vecteur. Néanmoins, cela a peu d'importance dans la mesure où il est facile, à partir de l'information des usines ouvertes, de retrouver la solution minimisant le coût total. En effet, il suffit de placer chaque client dans l'usine la plus proche (*i.e.* au plus faible coût d'affectation) parmi celles ouvertes. L'algorithme correspondant est l'Algorithme 1.

Modéliser nos solutions par la liste des usines ouvertes permet ainsi :

- d'en évaluer le coût total, permettant de comparer les solutions entre elles,
- tout en n'allouant que M éléments.

C'est donc ce modèle qui sera utilisé par la suite. On notera que le terme de "solution" désigneradès lors tout vecteur binaire $s \in \{0,1\}^M$ tel que le j^{eme} élément est 1 si et seulement si l'usine j est ouverte. De plus, on pourra par commodité parler de la "liste des usines ouvertes" pour désigner ce vecteur.

Algorithm 1: Évaluation d'une solution

Entrée: Un vecteur binaire $s \in \{0,1\}^M$, indiquant quelles usines sont ouvertes

Sortie : Le coût total $c \in \mathbb{N}$ de la meilleure solution de répartir les N clients parmi les usines ouvertes b.

```
1 c ← 0
2 foreach client i do
3 |   Trouver l'usine j* qui minimise P_ij parmi les usines ouvertes s
4 |   c ← c + P_ij*
5 end
6 c ← c + s.P // On ajoute au coût total la somme des prix
      d'ouverture des usines ouvertes, résultat du produit s.P
7
8 return c
```

2.2 Première heuristique

Maintenant que l'on a choisi comment modéliser les solutions du problème, on peut écrire une heuristique fournissant une première solution, à coup sûr sous-optimale, mais raisonnable et servant de bon point de départ pour les recherches d'améliorations à suivre. Notre première approche a été gloutonne : prendre localement la meilleure décision possible, en espérant un bon résultat global. Pour ce faire, on affecte chaque client à l'usine la plus proche, *i.e.* minimisant la somme des prix d'ouverture et d'affectation. À noter qu'une fois une usine ouverte, son prix d'ouverture sera de 0 pour les prochains clients. L'algorithme correspondant est l'Algorithme 2.

Algorithm 2: Première ouverture gloutonne d'usines

Entrée: Une instance du problème de localisation discrète (*i.e.* les prix d'ouverture $(P_j)_{1 \leq j \leq M}$ et d'affectations $(P_{ij})_{1 \leq j \leq N,\ 1 \leq j \leq M}$)

Sortie : Une solution au problème, *i.e.* un vecteur $s \in \{0,1\}^M$, indiquant quelles usines ont été ouvertes

On crée une copie P' du vecteur des prix d'ouverture des usines
$s \leftarrow \mathbf{0}_M$ `// usines initialement fermées`
foreach *client* i **do**
    Trouver l'usine j^* qui minimise $P'_j + P_{ij}$ parmi les M usines
    $s[j^*] \leftarrow 1$ `// on indique l'usine` j^* `comme ouverte`

    $P'_{j^*} \leftarrow 0$ `// on met le prix d'ouverture de` j^* `à 0`

end
return s

Cet algorithme fournit ainsi une première solution. Malheureusement, elle est sensible à l'ordre (arbitraire) dans lequel sont parcourus les clients. En effet, on peut imaginer que le choix d'usine pour le 1^{er} client aurait pu être différent si les usines finalement ouvertes l'aient été d'emblée. On peut s'en convaincre avec l'instance à 2 clients et 2 usines suivante :

$$P = \begin{pmatrix} 10 & 1 \end{pmatrix} \quad \text{et} \quad A = \begin{pmatrix} 1 & 5 \\ 1 & 15 \end{pmatrix}$$

Appliquons notre algorithme glouton en traitant d'abord le 1^{er} client, puis le 2^{eme} :

1. Pour le client 1, on a le choix entre le placer :
    - dans l'usine 1, pour $10 + 1 = 11$,
    - ou dans l'usine 2, pour $1 + 5 = 6$.

    On le place donc dans l'usine 2, qui est à présent ouverte.

2. Pour le client 2, on a le choix entre le placer :
    - dans l'usine 1, pour $10 + 1 = 11$,
    - ou dans l'usine 2, pour $0 + 15 = 15$.

    On le place donc dans l'usine 1.

Coût total : $6 + 11 = 17$, avec les deux usines ouvertes. Mais si, puisque les deux seront finalement ouvertes, on commence par les ouvrir (pour un prix de $10 + 1 = 11$) avant de décider des affectations des clients, on remarque alors que :

- pour le 1^{er} client, on a le choix entre la 1^{ere} usine pour 1 et la 2^{eme} pour 5,
- pour le 2^{eme} client, on a le choix entre la 1^{ere} usine pour 1 et la 2^{eme} pour 15,

soit un coût minimum possible de $11 + 1 + 1 = 13$.

On peut donc améliorer notre heuristique en ajoutant à l'ouverture gloutonne initiale une réallocation similaire à celle effectuée dans notre algorithme d'évaluation :

1. d'abord, on ouvre des usines grâce à notre ouverture gloutonne,
2. puis on replace les clients au sein de ces usines ouvertes pour minimiser les prix d'affectations.

À noter qu'il est possible qu'après réallocation, certaines usines ouvertes lors de l'ouverture gloutonne ne soient finalement plus utilisées. C'est le cas dans l'exemple ci-dessus : l'ouverture gloutonne a ouvert les deux usines, mais la réallocation a placé les deux clients dans la 1^{ere} usine. On peut donc ajouter une troisième étape :

3. ne pas ouvrir les usines finalement non utilisées, et ainsi économiser leur coût d'ouverture.

L'algorithme de cette heuristique ainsi décrite est l'Algorithme 3.

Algorithm 3: Heuristique initiale

Entrée: Une instance du problème de localisation discrète (*i.e.* les prix d'ouverture $(P_j)_{1\leq j\leq M}$ et d'affectations $(P_{ij})_{1\leq j\leq N,\ 1\leq j\leq M}$)
Sortie : Une solution au problème, *i.e.* un vecteur $s' \in \{0,1\}^M$, indiquant quelles usines ont été ouvertes

```
1  s ← ouverture_gloutonne (P, A) // on ouvre des usines via
       l'Algorithme 2 page 6
2
3  s' ← 0_M // on n'ouvrira que les usines effectivement
       utilisées
4
5  foreach client i do
6  |   Trouver l'usine j* qui minimise P_ij parmi les usines ouvertes s
7  |   s'[j*] ← 1 // on ouvre effectivement j*
8  |
9  end
10 return s'
```

3 Voisinages des solutions

Notre heuristique décrite par l'Algorithme 3 fournit une première solution au problème de localisation discrète sous forme d'une liste d'usines ouvertes (le vecteur $s \in \{0,1\}^M$). Pour tenter d'améliorer cette solution, on est encouragé à en explorer un (ou plusieurs) voisinage(s).

3.1 Différents voisinages

Puisque nos solutions sont modélisées par des vecteurs binaires, il est possible de reprendre les boules de Hamming vues en cours : pour une solution $s \in \{0,1\}^M$ donnée, la boule de Hamming de taille k est l'ensemble des solutions obtenues en inversant k éléments. Tout vecteur solution s étant de taille M, pour tout k, la boule de Hamming de taille k contient donc $\binom{M}{k}$ voisins de s.

Prendre un voisin d'une solution $s \in \{0,1\}^M$ dans une boule de Hamming (disons de taille 1) revient à inverser un des éléments de s. Ainsi, changer un 0 en 1 s'interprète comme ouvrir une usine initialement (en considérant s comment point initial) fermée, et à l'inverse changer un 1 en 0 revient à fermer une usine initialement ouverte. On peut alors considérer deux autres voisinages, tous deux inclus dans la boule de Hamming de taille 1, en n'autorisant que des ouvertures (respectivement fermetures) d'usines.

3.2 Descente de voisinages

Avec un voisinage clairement défini, il est alors possible d'effectuer une "descente de voisinage". C'est un procédé par récurrence :

- on part d'une solution initiale s_0,
- et à chaque itération, on évalue chaque voisin de la solution courante s, et on sélectionne comme nouvelle solution courante le voisin minimisant la fonction objectif (*i.e.* le coût total).

Une condition d'arrêt évidente est d'arrêter le procédé si aucun voisin n'améliore le coût total. On a alors atteint un minimum local, et on retourne la meilleure solution rencontrée. Cette seule condition d'arrêt suffit à garantir la terminaison du procédé, puisque notre problème de localisation discrète admet a un nombre fini de solutions (puisqu'une solution est un vecteur de $\{0,1\}^M$, on a 2^M solutions possibles). Néanmoins, parcourir 2^M solutions (dans le pire des cas du point de vue du temps de calcul) peut être coûteux. On peut alors ajouter une seconde condition d'arrêt telle qu'un nombre maximal d'itérations. Le procédé ainsi décrit donne lieu à l'Algorithme 4.

Algorithm 4: Descente de voisinage

Entrée: Une instance du problème de localisation discrète (*i.e.* les prix d'ouverture $(P_j)_{1 \leq j \leq M}$ et d'affectations $(P_{ij})_{1 \leq j \leq N,\ 1 \leq j \leq M}$),
Entrée: une solution initiale $s_0 \in \{0,1\}^M$,
Entrée: un voisinage $V : \{0,1\}^M \to \{0,1\}^M$,
Entrée: et un nombre d'itérations maximal $iterMax \in \mathbb{N}$.
Sortie : Une solution au problème, *i.e.* un vecteur $s' \in \{0,1\}^M$, indiquant quelles usines ont été ouvertes

```
1  s ← s_0
2  compteurIterations ← 0
3  stop ← Faux
4  while ¬ stop ∧ compteurIterations < iterMax do
5  |  Trouver le voisin v* au plus faible coût total dans le voisinage de
   |   V(s)
6  |  if evaluation(v*) > evaluation(s) then
   |  |  // RQ : evaluation est l'algo 1 page 5
7  |  |  stop ← Vrai
8  |  end
9  |  else
10 |  |  s ← v*
11 |  |  compteurIterations ← compteurIterations + 1
12 |  end
13 end
14 return s
```

4 Métaheuristiques

À toute instance donnée, notre heuristique initiale (Algorithme 3 page 7) nous fournit une première solution, et notre descente de voisinage (Algorithme 4 page 9) permet de l'améliorer jusqu'à un minimum local. Pour continuer d'améliorer notre solution, nous avons alors implémenté trois métaheuristiques classiques : le recuit simulé, la recherche tabou et un algorithme génétique. Nous présentons dans cette Section le paramétrage retenu pour chacune de ces approches, après avoir testés de nombreuses configurations sur les instances fournies.

4.1 Recuit simulé

Le recuit simulé a été implémenté tel que présenté en cours, et est décrit par l'Algorithme 5.

Un recuit simulé prend de nombreux paramètres en argument :

- une solution initiale,
- une structure de voisinage à explorer à chaque itération,
- la température initiale du système,
- une condition d'arrêt de l'algorithme,
- un nombre de voisins à explorer à chaque température,
- et un coefficient μ pour faire décroître la température géométriquement ($T \leftarrow \mu.T$).

Tous ces paramètres sont autant de réglages à ajuster pour tenter d'améliorer l'efficacité du recuit. Nous avons ainsi effectué de nombreux tests pour déterminer quelles valeurs fournissaient la meilleure solution. De ces tests, le constat est simple (et attendu) : plus le nombre de solutions visitées par le système est grand, meilleur est le résultat, mais plus long est le temps de calcul. Voici notamment les résultats obtenus en faisant varier la température initiale et le coefficient μ (avec pour solution initiale la solution fournie par l'algorithme 2, comme condition d'arrêt un seuil de température à 10^{-3}, et 10 voisins explorés par température, pris dans la Boule de Hamming de taille 1 car c'est celle qui fournissait les meilleures solutions) :

- la Figure 1 montre visuellement qu'on a de meilleures solutions en prenant la température initiale la plus élevée possible, et μ le plus proche de 1,
- la Figure 2 confirme ce constat pour μ,
- mais la Figure 3 montre bien l'explosion du temps de calcul quand μ tend justement vers 1.

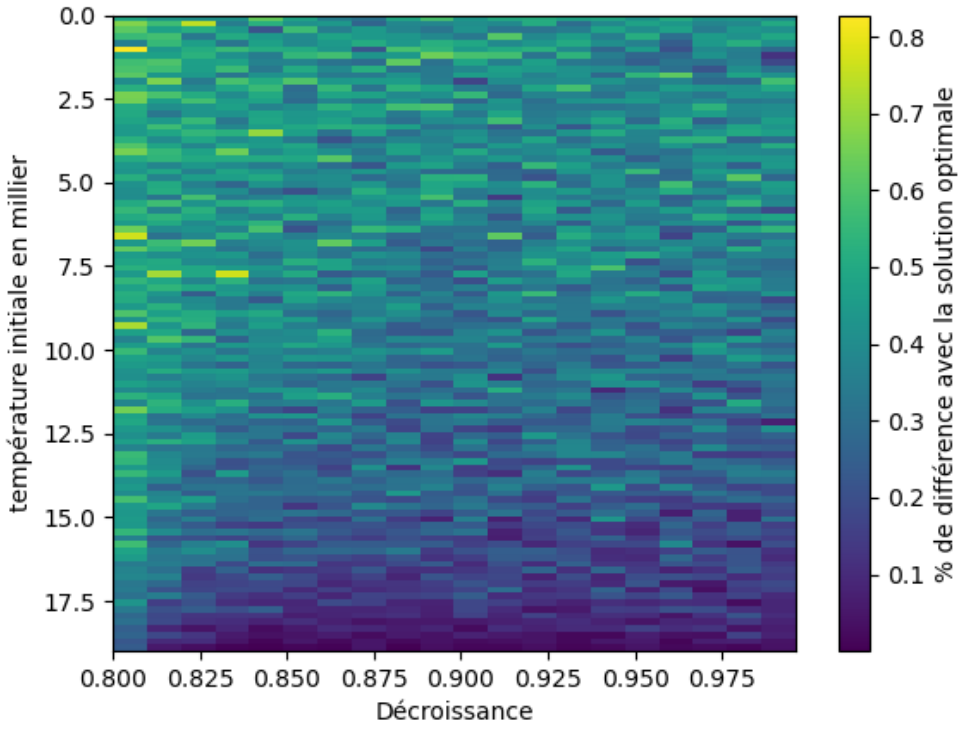


Figure 1: Influence de T_{init} et de μ sur le recuit simulé

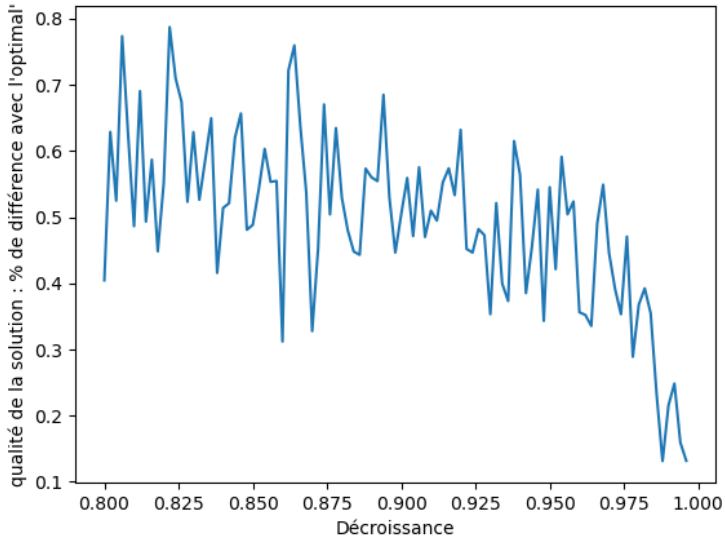


Figure 2: Qualité de la solution en fonction de μ

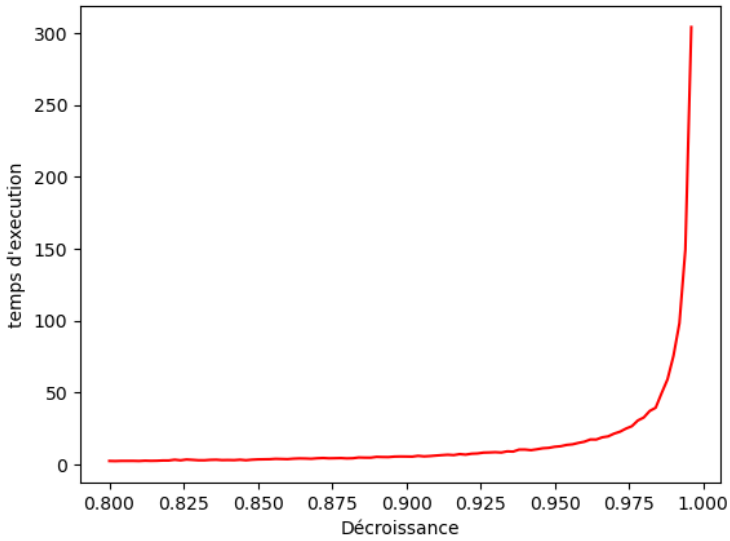


Figure 3: Temps d'exécution du recuit simulé en fonction de μ

Algorithm 5: Recuit simulé

Entrée: Une instance du problème de localisation discrète (*i.e.* les prix d'ouverture $(P_j)_{1\leq j\leq M}$ et d'affectations $(P_{ij})_{1\leq j\leq N,\ 1\leq j\leq M}$),
Entrée: une solution initiale $s_0 \in \{0,1\}^M$,
Entrée: un voisinage $V : \{0,1\}^M \to \{0,1\}^M$,
Entrée: la température initiale $T_{init} \in \mathbb{R}$,
Entrée: le seuil de température pour la condition d'arrêt $T_{seuil} \in \mathbb{R}$,
Entrée: le nombre de voisins à explorer par température $n_{voisins} \in \mathbb{N}$,
Entrée: et un coefficient de décroissance géométrique de la température $\mu \in \mathbb{R}$.
Sortie : Une solution au problème, *i.e.* un vecteur $s' \in \{0,1\}^M$, indiquant quelles usines ont été ouvertes

```
1  s_min ← s_0 // meilleure solution visitée
2
3  s ← s_0 // solution courante
4
5  T ← T_init
6  while T > T_seuil do
7  |  for i = 0 to n_voisins do
8  |  |  Tirer au sort un voisin v dans V(s)
9  |  |  Δ ← evaluation(v) - evaluation(s)
10 |  |  if Δ ≤ 0 then
11 |  |  |  s ← v
12 |  |  |  if evaluation(s) < evaluation(s_min) then
13 |  |  |  |  s_min ← s
14 |  |  |  end
15 |  |  end
16 |  |  else
17 |  |  |  Tirer au sort q dans [0,1]
18 |  |  |  if q < exp(-Δ/T) then
19 |  |  |  |  s ← v
20 |  |  |  end
21 |  |  end
22 |  end
23 |  T ← μT
24 end
25 return s_min
```

Remarque : pour un petit nombre d'usine M, à chaque itération on peut explorer le voisinage entier de la solution s. Mais pour M grand, il est plus sûr de contrôler le nombre de voisins explorés en en tirant $n_{voisins}$ au sort.

4.2 Recherche tabou

Comme pour le recuit simulé, la recherche tabou a été implémentée telle que présentée en cours, et est décrit par l'Algorithme 6.

Une recherche tabou prend deux paramètres en argument (sans compter la solution intiale, fournie ici aussi par l'algorithme 2) :

- la taille de la liste tabou,
- et une condition d'arrêt.

Après avoir testés différentes valeurs, nous avons retenu une taille maximale pour notre liste tabou égale à la moitié du nombre d'usines. Pour la condition d'arrêt, nous avons opté pour un temps (nombre d'itérations) limite que l'on accepte d'attendre avant de trouver une meilleure solution que la précédente. La Figure 4 montre que sur les instances fournies, même un petit temps d'attente suffit à trouver de très bonnes solutions. En revanche, puisque le temps de calcul, lui, a une croissance linéaire, on retiendra un temps limite de 10 itérations.

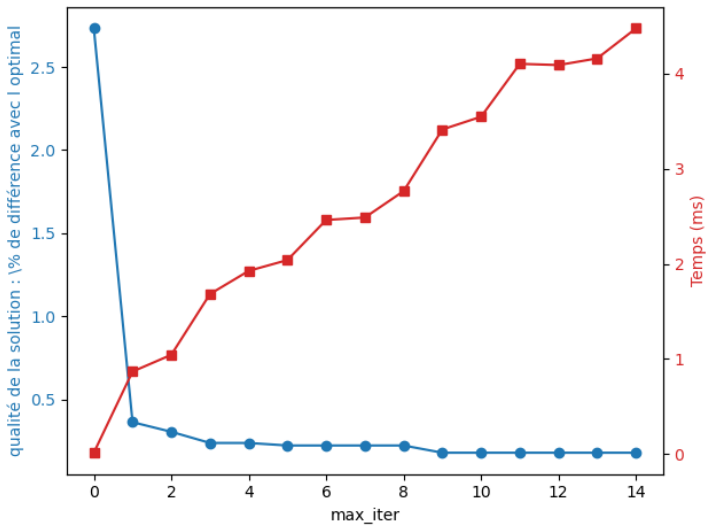


Figure 4: Performance et temps d'exécution selon max_iter

Algorithm 6: Recherche tabou

Entrée: Une instance du problème de localisation discrète (*i.e.* les prix d'ouverture $(P_j)_{1 \leq j \leq M}$ et d'affectations $(P_{ij})_{1 \leq j \leq N,\ 1 \leq j \leq M}$),
Entrée: en critère d'arrêt : un nombre maximal d'itérations passées sans améliorer la solution, $n_{stuck} \in \mathbb{N}$.
Sortie : Une solution au problème, *i.e.* un vecteur $s' \in \{0,1\}^M$, indiquant quelles usines ont été ouvertes

$s_{min} \leftarrow s_0$ `// meilleure solution visitée`

$s \leftarrow s_0$ `// solution courante`

$n \leftarrow 0$ `// un compteur pour les itérations sans améliorations`

taboo_list $\leftarrow$ [] `// liste tabou initialement vide`

while $n < n_{stuck}$ **do**
    best_voisin $\leftarrow$ un voisin de s
    for $i = 0$ *to* $n_{voisins}$ **do**
        Tirer au sort un voisin v dans $V(s)$
        if v *non tabou* $\wedge$ *evaluation(v)* $<$ *evaluation(s)* **then**
            best_voisin $\leftarrow v$
        end
        if v *tabou* $\wedge$ *evaluation(v)* $<$ *evaluation*(s_{min}) **then**
            best_voisin $\leftarrow v$
        end
    end
    $s \leftarrow$ best_voisin `// on actualise` s `même si cela dégrade la solution`

    if *evaluation(s)* $<$ *evaluation*(s_{min}) **then**
        $s_{min} \leftarrow s$
    end
    Ajouter best_voisin à taboo_list
end
return s_{min}

Remarque : la nature de l'information stockée dans la liste tabou dépend du voisinage considéré. Pour la boule de Hamming de taille 1 par exemple, il suffit de stocker l'indice de l'élément de $s \in \{0,1\}^M$ inversé pour obtenir $v \in \{0,1\}^M$ (*i.e.* l'usine qui a été ouverte/fermée).

4.3 Algorithme génétique

L'algorithme génétique présenté repose sur une approche évolutive classique, on a une phase de descente locale et d'un mécanisme de sélection inspiré d'un tournoi compétitif. Voici son fonctionnement :

On génère un ensemble de solutions candidates stocké dans le vecteur `population`. Chaque élément représente une solution du problème.

Chaque individu de la population initiale subit une descente locale afin d'affiner la qualité des solutions avant la reproduction. Les solutions améliorées sont stockées dans le vecteur `conclave_parent`. Cette étape vise à introduire une exploitation locale des minima, en complément de l'exploration globale assurée par les croisements.

À partir des parents du `conclave_parent`, on forme des couples qui engendrent deux enfants chacun. Le croisement se fait de manière à conserver l'intégralité de l'information génétique :

Parents	Enfants
AB	Ab
ab	aB

Ce schéma garantit que chaque gène des parents est transmis à au moins un descendant, assurant ainsi une diversité contrôlée tout en évitant la perte d'information utile. Les enfants ainsi générés sont placés dans le vecteur `conclave_enfant`.

Les vecteurs `conclave_parent` et `conclave_enfant` contiennent à présent l'ensemble des solutions (parents et enfants) accompagnées de leur coût. Les deux ensembles sont fusionnés puis triés selon leur performance. Les meilleures solutions issues de cette confrontation (d'où la désignation de *conclave*, représentant un tournoi) sont sélectionnées pour former la nouvelle population.

La nouvelle population remplace l'ancienne et le processus est répété sur un nombre défini de générations. Deux paramètres principaux gouvernent donc l'évolution :

- la taille de la population, qui détermine la diversité génétique;
- le nombre de générations, qui conditionne la profondeur de la recherche.

Un compromis est à trouver entre ces deux facteurs :

- une grande population et peu de générations favorisent une exploration large mais superficielle, comparable à une pluie de billes ;
- à l'inverse, une population réduite sur de nombreuses générations renforce la convergence locale mais risque de limiter la diversité.

Choix des Paramètres :

On remarque que le temps d'excecution est proportionnel à la taille de la population. Augmenter le nombre de génération augmente moins fortement le temps d'excecution autour de une seconde

nombre de génération	**taille de la population**	**temps d'excecution**	**Qualité de la solution en % d'écart**
10	10	596	0.0290
10	20	1196	0.0037
10	50	2976	0.0000000198
10	100	5932	0.0000000198
50	10	1849	0.0136
100	10	3355	0.0225
5	24	1052	0.0032
10	18	1070	0.0056
15	14	1045	0.0034967552
20	12	1082	0.0129
25	10	1061	0.0129186004
50	6	1089	0.0484

Table 1: Tableau comparatif des performances pour l'algorithme génétique

Dans la deuxième partie du tableau, nous avons testé plusieurs couple de (nombre de génération,taille de la population) de tel sorte à garder le temps proche de une seconde.

A temps équivalent (1 seconde), il est ainsi préférable d'avoir plus de population que de génération. Cependant, avec cette approche, l'algorithme se rapproche de la Pluie de bille (que nous expliquerons par la suite). Par la suite, nous prendrons donc une population de 10 individus afin de différentier ces deux méthodes.

4.4 Pluie de bille

Pendant l'implémentation des différentes métaheuristiques, nous avons souhaité tester (à nouveau) notre descente de voisinage (présenté Section 3.2), mais en en réalisant plusieurs simultanément, à partir de plusieurs solutions initiales générées aléatoirement.

Algorithm 7: Pluie de billes

Entrée: Une instance du problème de localisation discrète (*i.e.* les prix d'ouverture $(P_j)_{1 \leq j \leq M}$ et d'affectations $(P_{ij})_{1 \leq j \leq N,\ 1 \leq j \leq M}$),
Entrée: le nombre $n_{descente}$ de descentes souhaité
Sortie : Une solution au problème, *i.e.* un vecteur $s' \in \{0,1\}^M$, indiquant quelles usines ont été ouvertes

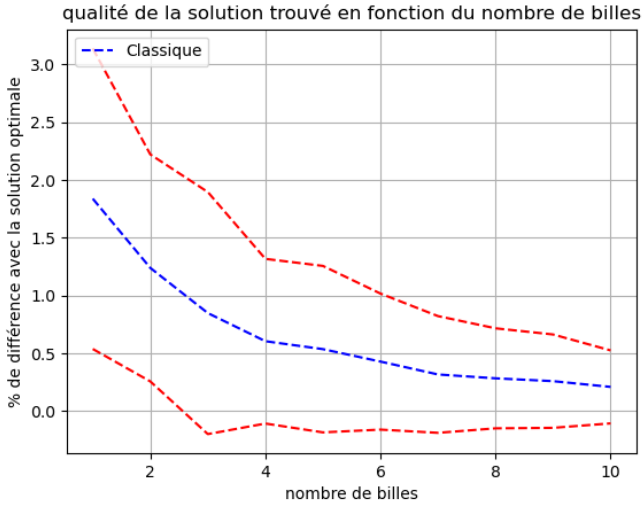
```
1 s* ← une solution tirée aléatoirement
2 for i = 0 to n_descente do
3 |   s_0 ← une solution tirée aléatoirement
4 |   Effectuer une descente de voisinage sur s_0
5 |   if evaluation(s_0) < evaluation(s*) then
  |   |   // RQ : evaluation est l'algo 1 page 5
6 |   end
7 |   s* ← s_0
8 end
9 return s*
```

Cette heuristique très simple s'avère étonnamment efficace. Cette forte performance s'explique par le fait qu'elle visite beaucoup de solution à la seconde, puisqu'elle ne fait que ça (hormis l'initialisation aléatoire).

Remarque : la dénomination vient du fait que l'on a des solutions répartie aléatoirement, comme des gouttes d'eau. Elles arrivent ensuite sur le sol, dont la topologie représente le coût de la solution. Finalement, la bille glisse sur le sol et descend, ce qui représente la descente locale.

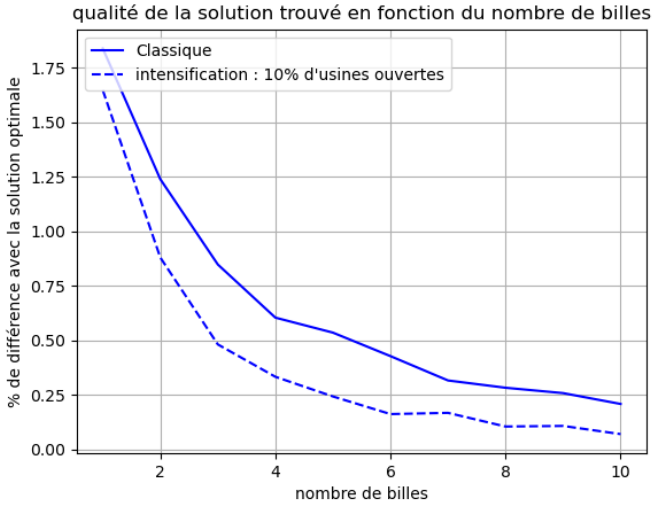
De plus, en analysant les solutions retournées, nous avons remarqué que pour la plupart des instances la meilleure solution trouvée contient peu d'usines ouvertes. Nous avons alors eu l'idée d'exploiter ce constat lors de la génération aléatoire de nos solutions aléatoires (ligne). Ainsi, dans l'algorithme obtenu, dit de la "Buine de billes", la probabilité d'avoir un 1 est non pas de 0.5, mais de 0.1, et l'on commence donc avec des solutions contenant seulement 10% d'usines ouvertes (processus d'intensification).

Remarque : cette nouvelle dénomination fait référence à ces vecteurs composés de peu de 1, donc moins "lourds" et formant ainsi de plus petites gouttes.



Pour les figures suivantes, nous avons restreint l'échantillon de test aux quatre dernières instances, de plus grande taille que les précédentes, afin d'éviter des valeurs trop faibles. Les résultats présentés correspondent donc à la somme des erreurs ou des temps obtenus sur ces quatre instances. Étant donné la part de hasard dans la méthode, nous avons répété chaque expérience 50 fois et calculé la moyenne des résultats. Les courbes rouges représentent les intervalles de confiance à ± 2 écarts-types.

En comparaison, avec la *Pluie de billes* :



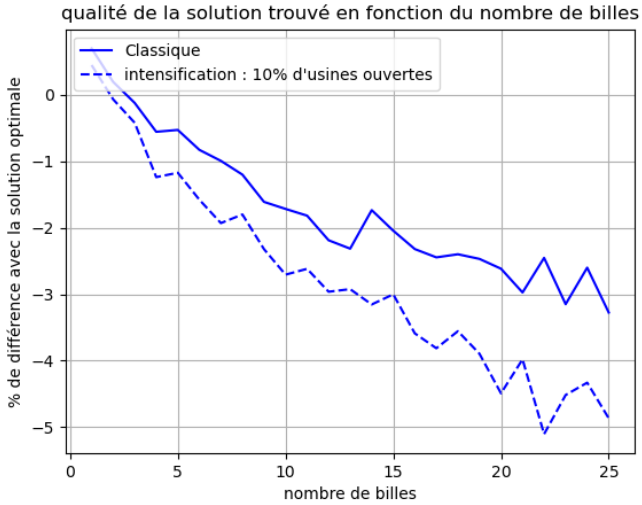
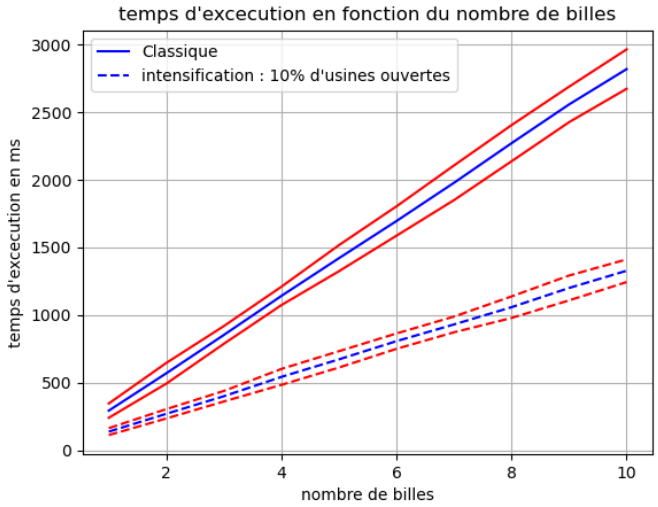


Figure 5: Log de la qualité de la solution en fonction du nombre de billes

On observe que cette méthode est nettement plus efficace pour un même nombre de billes. Mais l'avantage ne s'arrête pas là : elle est également plus rapide. Cela s'explique par le fait qu'évaluer une solution contenant peu de 1 (donc peu d'usines ouvertes) demande moins de calculs. 9

De plus, le graphique semble indiquer une relation logarithmique entre le nombre de billes et la qualité de la solution.



Ainsi, ce double avantage (en qualité et en rapidité) permet à la *Bruine* d'être nettement supérieure à la Pluie pour un même temps imparti.

Cependant, deux questions se posent :

1. Pourquoi choisir 10 % ?
2. Que se passe-t-il si les bonnes solutions nécessitent un grand nombre d'usines ouvertes ?

En effet, en commençant avec seulement 10 % d'usines ouvertes, on parie que les meilleures solutions présenteront elles aussi un faible taux d'ouverture. Or, si les coûts d'ouverture sont très faibles par rapport aux coûts d'affectation, il devient préférable d'ouvrir presque toutes les usines ; dans ce cas, la *Bruine* risque d'explorer de nombreuses solutions de mauvaise qualité.

Pour pallier ce problème, nous avons développé une version adaptative de la *Bruine*. Cette version calcule le ratio

$$\text{Ratio} = \frac{\text{Coût d'ouverture}}{\text{Coût d'affectation}}$$

et utilise une table de correspondance pour déterminer le pourcentage optimal d'usines à ouvrir lors de l'initialisation.

Concrètement, nous avons pris les quatre dernières instances, puis normalisé les coûts d'ouverture et d'affectation de sorte que leur somme soit égale à 1. Ainsi, pour un ratio égal à 1, les deux types de coûts ont le même poids. Pour obtenir un ratio arbitraire, nous multiplions les coûts d'ouverture par ce ratio.

Pour chaque valeur de ratio, nous générons un ensemble de solutions possédant différents pourcentages d'usines ouvertes, uniformément répartis entre 0 % et 100 %, afin de couvrir toutes les configurations possibles. Le graphique suivant illustre cette relation :

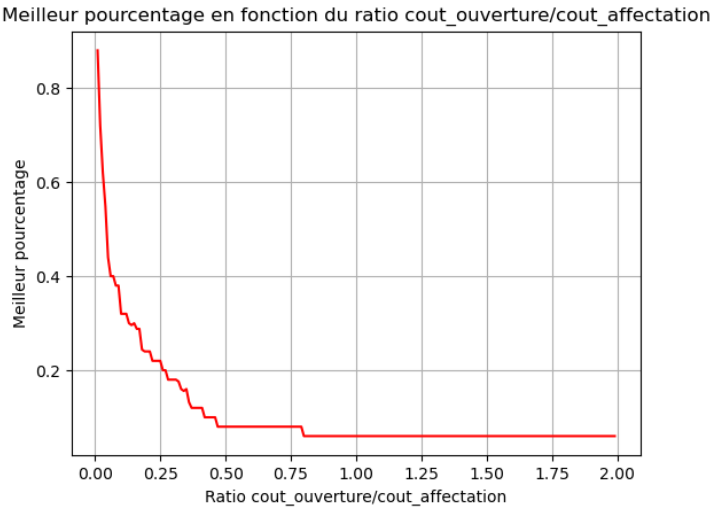


Figure 6: Meilleur pourcentage en fonction du ratio (coût ouverture/ coût affectation)

On confirme ainsi l'hypothèse selon laquelle un ratio faible (coûts d'ouverture faibles) conduit à un faible pourcentage d'usines ouvertes dans les meilleures solutions.

5 Résultats

Nous présentons dans cette Section les résultats obtenus lors des évaluations des performances de nos différents algorithmes. Quelques précisions importantes :

- Chaque résultat présenté (que ce soit la qualité de la solution ou son temps de calcul) est une moyenne sur les instances fournies,
- Les résultats des algorithmes faisant intervenir une part de hasard (pluie de billes, recuit simulé *etc*) sont eux-même une moyenne d'au moins 30 exécutions (pour chaque instance),
- La qualité de la solution est présentée sous forme d'un pourcentage d'erreur par rapport à la valeur optimale fournie (pour chaque instance), et a été calculée selon la formule suivante :

$$q = (1 - \frac{\text{coût total de la solution trouvée}}{\text{coût optimal fourni}}) * 100$$

- Pour que les résultats soient comparables d'une heuristique à l'autre, ils ont été "normalisés" : les paramètres ont été choisis de telle sorte que l'appel à chaque heuristique prenne environ 1 seconde. Par exemple, c'est avec un coefficient de décroissance $\mu = 0.993$ que l'appel au recuit simulé prenait 1 seconde à atteindre sa condition d'arrêt sur notre machine.
- Néanmoins, cette "normalisation" n'a pas pu être effectuée sur la recherche tabou, qui ne dépassait jamais les 300 millisecondes quelque soient les paramètres donnés.

Nom	Temps en ms	Coût moyen trouvé	% d'écart à l'optimal
Recherche Tabou	1.5	910182	0.02375
Recuit ($\mu = 0.999$)	944	908153	0.0076
Algo Génétique	956	908207	0.0144
Génétique (5 générations))	929	908113	0.0030
Pluie de billes	949	908105	0.0020
Bruine de billes (10%)	954	908091	0.00028
Bruine de billes (5%)	959	908093	0.00049
Bruine de billes adaptative	947	908091	0.00030

Table 2: Comparatif des performances sur les "petites" instances

Nom	Tempe en s	Coût moyen trouvé	% d'écart à l'optimal
Recuit ($\mu = 0.999$)	34	15 059 272	8.2340
Pluie de bille	46	14 165 663	2.3201
Bruine de bille	56	14 028 503	1.0650
Bruine de bille adaptative	54	14 055 352	1.2295

Table 3: Comparatif des performances sur les 3 "grosses" instances (capa.txt, capb.txt et capc.txt)

On constate ainsi que c'est notre heuristique de la bruine de billes (paramétrée à 10% d'ouverture) qui fournit les meilleurs résultats, et ce malgré sa grande simplicité. Notre interprétation est que l'espace des solutions est suffisamment petit pour être largement exploré en générant beaucoup de solutions aléatoires. Ceci est appuyé par le constat que notre recuit simulé (qui lui aussi présente une grande part d'aléatoire) est très peu sensible au choix de la solution initiale : avec une température initiale très élevée et une décroissance très lente, le recuit peut partir très "loin" de la solution initiale.

Remarque : tous les algorithmes n'ont pas été autant testés sur les 3 grosses instances que sur les plus petites, en raison de leur temps d'exécution. Néanmoins, l'écart de performances entre notre bruine de billes et le recuit simulé nous laisse penser que là encore, la bruine semble fournir les meilleures solutions.

6 Version difficile du problème

Dans cette Section, on considère la version "difficile" du problème de localisation discrète : les usines n'ont maintenant plus une capacité d'accueil illimitée. Chaque client a une demande, et chaque usine une capacité finie. En plus des prix d'ouverture P et d'affectation A, toute instance englobe désormais également les vecteurs de demandes $D \in \mathbb{R}^N$ et de capacités $C \in \mathbb{R}^M$.

6.1 Notion de faisabilité d'une solution

Avec l'introduction des demandes et capacités, certaines répartitions des N clients entre les M usines, considérées jusqu'à présent comme des solutions au problème "simple", deviennent irréalisables. Il suffit pour cela que la capacité d'une seule usine soit dépassée par les demandes des clients qu'elle accueille. La question est alors de savoir à quel point cette notion de faisabilité d'une solution perturbe notre minimisation.

6.2 Approche naïve

Une première approche consiste alors à se dire que certes, certaines répartitions seront irréalisables, mais il suffit de les exclure.

Avec une fonction testant la faisabilité d'une solution (Algorithme 8), on peut en effet tenter de réappliquer les mêmes recherches que face au problème "simple", mais en excluant toute solution irréalisable. Il suffit de partir d'une solution initiale réalisable, et d'appliquer exactement les mêmes algorithmes que précédemment (descente de voisinage, recuit simulé, *etc.*) en se contentant de n'explorer que les voisins réalisables au sein de nos voisinages.

Formellement, pour une solution s donnée, si l'on considère un voisinage V parmi ceux donnés à la Section 3.1 (par exemple la boule de Hamming de taille 1), on peut définir un nouveau voisinage V' adapté à la version difficile du problème par :

$$V' = V \setminus \{v \in V, \neg\text{est\_faisable}(v)\}$$

Cette approche, d'apparence naïve, est tout à fait envisageable pour les instances peu restrictives, *i.e.* dont la majorité des répartitions ne violent aucune contrainte. On remarquera que c'est notamment le cas des instances fournies pour ce projet. Elles ont en effet été construites de telle sorte que la capacité de chaque usine vaille précisément la somme des demandes de tous les clients, garantissant ainsi que toute répartition soit réalisable (même placer tous les clients dans une seule usine, celle de plus petite capacité).

Algorithm 8: Test de la faisabilité d'une répartition

Entrée: Une instance du problème difficile, *i.e.* une matrice A et les vecteurs P, C et D.

Entrée: Une répartition des clients dans les usines, représentée par une matrice binaire $M \in \mathcal{M}_{N,M}(\{0,1\})$ indiquant chaque affectation.

Sortie : Un booléen indiquant si M est réalisable sans violer la moindre contrainte de capacité.

```
1 viol ← False
2 foreach usine j do
3 |   occupation_j ← somme des demandes des clients affectés à j
4 |   if occupation_j > C_j then
5 |   |   viol ← True
6 |   end
7 end
8 return viol
```

6.3 Cas des instances très restrictives

Si l'approche décrite précédemment fonctionne sur des instances peu restrictives, elle est peu robuste et échoue face à des instances plus contraignantes.

Prenons l'exemple du recuit simulé (Algorithme 5 page 13). À chaque itération, on tire au hasard des voisins de la solution courante s. Mais alors si la majorité des solutions sont non réalisables (globalement, ou ne serait-ce que dans la région de l'espace des solutions en cours d'exploration locale), un grand nombre de tirages sont bloquants.

Et en réalité, on peut rencontrer un problème bien avant d'en arriver au recuit simulé : rien ne dit que notre "solution" telle que modélisée depuis la Section 2.1 soit réalisable. En effet, l'idée exploitée dans l'Algorithme 1 était qu'à partir d'une liste d'usines d'ouvertes on pouvait toujours trouver la répartition des clients parmi ces usines ouvertes permettant de minimiser le coût total. Mais rien ne garantit la faisabilité de cette répartition. Au contraire, face à des instances particulièrement contraignantes, on risque d'avoir quasi-systématiquement nos "solutions" (au sens de la répartition obtenue par l'Algorithme 1) non réalisables - ce qui bien-sûr paralyse la recherche.

Ainsi, mettre en place une approche résistante face aux instances les plus contraignantes nécessite de renoncer à réagir aux viols de contrainte de façon binaire (ou booléenne), en conservant ou excluant une solution. Pour se déplacer dans un espace de solutions où l'écrasante majorité des solutions sont irréalisables, il faut quantifier leur viol de contraintes. L'approche du problème consiste alors à considérer deux objectifs à minimiser (et non plus un seul) : le coût total et la quantité de viol de contraintes.

On avait déjà une fonction d'évaluation du coût total d'une solution (Algorithme 1), l'Algorithme 9 (variante du test de faisabilité précédent) fournit à présent une fonction d'évaluation du viol total d'une solution.

Algorithm 9: Évaluation du viol total d'une solution

Entrée: Une instance du problème difficile, *i.e.* une matrice A et les vecteurs P, C et D.
Entrée: Une répartition des clients dans les usines, représentée par une matrice binaire $M \in \mathcal{M}_{N,M}(\{0,1\})$ indiquant chaque affectation.
Sortie : Un booléen indiquant si M est réalisable sans violer la moindre contrainte de capacité.

```
1 viol_total ← 0
2 foreach usine j do
3 |   occupation_j ← somme des demandes des clients affectés à j
4 |   if occupation_j > C_j then
5 |   |   viol_total ← occupation_j − C_j
6 |   end
7 end
8 return viol_total
```

Avec deux critères à minimiser (les coût et viol totaux) et deux fonctions d'évaluation, il est alors possible de reprendre les approches basées sur une exploration locale vues face au problème "simple". Reste à décider à chaque étape si l'on souhaite partir dans une direction minimisant le viol ou le coût (ou peut-être de faire d'emblée un seul objectif fonction des deux). On peut par exemple opter pour une approche cherchant à se ramener le plus vite possible (*i.e.* quel que soit le coût, en choisissant systématiquement le voisin réduisant le plus le viol total) à une région de l'espace aux solutions réalisables, puis de minimiser le coût sans jamais en ressortir (Algorithme 10).

Algorithm 10: Descente de viol, puis de coût

Entrée: Une instance du problème difficile, *i.e.* une matrice A et les vecteurs P, C et D.
Entrée: une solution initiale $s_0 \in \{0,1\}^M$,
Entrée: un voisinage $V : \{0,1\}^M \to \{0,1\}^M$,
Entrée: et un nombre d'itérations maximal $iterMax \in \mathbb{N}$.
Sortie : Une solution au problème, *i.e.* un vecteur $s' \in \{0,1\}^M$, indiquant quelles usines ont été ouvertes

```
1  s ← s_0
2  compteurIterations ← 0
3  stop ← Faux
4  while ¬ stop ∧ compteurIterations < iterMax do
5  |  Trouver le voisin v* au plus faible viol total dans le voisinage de
   |   V(s)
6  |  if eval_viol(v*) > eval_viol(s) then
   |  |  // RQ : eval_viol est l'algo 9 page 27
7  |  |  stop ← Vrai
8  |  end
9  |  else
10 |  |  s ← v*
11 |  |  compteurIterations ← compteurIterations + 1
12 |  end
13 end
   // réinitialisation des paramètres pour la deuxième descente
14 compteurIterations ← 0
15 stop ← Faux
16 while ¬ stop ∧ compteurIterations < iterMax do
17 |  Trouver le voisin réalisable v* au plus faible coût total dans le
   |      voisinage V'(s) // V' a été défini à la Section 6.2
18 |
19 |  if eval_cout(v*) > eval_cout(s) then
   |  |  // RQ : eval_cout est l'algo 1 page 5
20 |  |  stop ← Vrai
21 |  end
22 |  else
23 |  |  s ← v*
24 |  |  compteurIterations ← compteurIterations + 1
25 |  end
26 end
27 return s
```

Néanmoins, cette "descente" est très catégorique (priorité absolue donnée au viol), et des combinaisons plus fines des deux critères fourniraient sans doute de meilleures solutions (*i.e.* des solutions réalisables aux coûts totaux plus faibles).

Ces approches plus fines n'ont malheureusement pas pu être explorées, puisque les instances fournies n'étaient (littéralement) pas restrictives : aucune répartition immaginable des N clients entre les M usines ne violait la moindre contrainte. Nous ajoutons néanmoins deux éléments qui pourraient s'avérer utiles face à des instances plus coriaces : un nouvel algorithme glouton et un nouveau voisinage, plus fin.

L'algorithme glouton 11 est au viol ce que l'ouverture gloutonne page 6 (Algorithme 2) est au coût : il fournit une solution initiale en plaçant les clients là où ils violent le moins de contraintes. Il est notamment tout indiqué pour fournir une solution initiale à la descente de viol ci-dessus (Algorithme 10).

Algorithm 11: Première ouverture gloutonne d'usines

Entrée: Une instance du problème difficile, *i.e.* une matrice A et les vecteurs P, C et D.
Sortie : Une matrice d'affectation M

```
1  M ← 0_{N,M}
2  On trie les clients par ordre décroissant de demande
3  On trie les usines par ordre décroissant de capacité
4  foreach client i pris par ordre décroissant de demande do
5  |  j ← première usine (à la plus grosse capacité)
6  |  if j a la place d'accueillir i then
7  |  |  M_ij ← 1 // on affecte i à j
8  |  |
9  |  |  C_j ← C_j − D_i // on actualise la capacité de j
10 |  |
11 |  end
12 |  else
13 |  |  j ← usine suivante (ordre décroissant des capacités)
14 |  end
15 end
16 return M
```

Enfin, on achève cette Section par une remarque. Dans la version "simple" du problème, on manipulait des listes d'ouverture d'usines en guise de solution. Prendre un voisin dans une boule de Hamming revenait alors à ouvrir ou fermer une usine. Dans la version "difficile" du problème, autant ouvrir une usine est toujours possible (dans le sens où certes cela peut augmenter le coût total, mais ça ne peut que soulager le viol total), fermer une usine est en revanche radical et peut mener à un énorme saut sur le viol total. Pour explorer des solutions à la variation de viol moins brusque, il peut être intéressant de considérer un nouveau voisinage. L'idée est de ne plus raisoner sur les listes d'ouverture d'usines, mais sur les matrices de répartition (présentées page 4) : l'élément en position (i, j) vaut *1* ssi le client i est placé dans l'usine j (0 sinon). Avec ces matrices, on ne peut pas reproduire directement les boules de Hamming : changer un 1 en 0 signifie qu'un client n'est plus placé, et changer un 0 en 1 donnerait deux 1 sur une même ligne, impliquant qu'un client est placé à deux usines distinctes. Néanmoins, on peut tout de même implémenter le voisinage intuitif de "déplacer un seul client". En effet, l'algorithme 12 associe à toute répartition S la liste de déplacements de clients possibles (*i.e.* sans violer de contrainte). Chaque déplacement fournit alors un voisin pour S.

Algorithm 12: Génération du voisinage de déplacement de clients

Entrée: Une instance du problème difficile, *i.e.* une matrice A et les vecteurs P, C et D.
Entrée: Une solution du problème difficile, *i.e.* une matrice $S \in \mathcal{M}_{N,M}(\{0,1\})$
Sortie : Une liste de déplacements de clients possibles à partir de S

```
1  res ← ([])_N // un vecteur de N listes initialement vides
2
3  foreach client i do
4  |  foreach usine j do
5  |  |  if S_ij = 0 ∧ C_j > D_i then
6  |  |  |  Ajouter j à res[i]
7  |  |  end
8  |  end
9  end
10 return res
```

7 Conclusion

Face au problème classique de la localisation discrète, nous avons imaginé, implémenté et testé différentes approches, allant de la plus simple heuristique gloutonne à de plus complexes métaheuristiques comme l'algorithme génétique. Après avoir réalisé différents tests de performances, il s'avère que chacune de nos approches fournit de bonnes solutions sur les différentes instances fournies. Néanmoins, l'une d'entre elle sort du lot : notre heuristique dite de la "bruine de billes". Sa victoire est quelque peu surprenante vue sa simplicité, et invite interprétation. Nous estimons qu'elle doit son efficacité à la place qu'elle laisse au hasard, permettant d'explorer beaucoup de régions différentes de l'espace des solutions, ce dernier n'étant pas bien grand pour les instances fournies.

Face cette fois-ci à la version "difficile" du problème (où les usines n'ont plus de capacité d'acceuil infinie), une réflexion a été menée selon le caractère restrictif de l'instance à traiter, avec différentes pistes d'approches possibles.