Recuit simulé multi-thread

Descormier Natello & Vielzeuf Charles

6 novembre 2025

1 Réponse aux questions

1.1 Heuristiques d'initialisation utilisées

Le solveur implémente trois heuristiques d'initialisation distinctes pour générer des solutions de départ. Ces méthodes construisent progressivement une solution en assignant chaque client à une installation, en respectant éventuellement les contraintes de capacité si requis.

1. Heuristique gloutonne (GreedyInit) Cette méthode utilise une approche déterministe optimale locale. Pour chaque client i, l'algorithme :

- Calcule le coût d'assignation à chaque installation j déjà ouverte : c_{ij}
- Calcule le coût total pour ouvrir une nouvelle installation j : $c_{ij} + f_j$
- Sélectionne l'option la moins coûteuse parmi toutes les installations faisables

La formule de décision est :

$$\text{coût\_total}(i,j) = \begin{cases} c_{ij} & \text{si l'installation } j \text{ est déjà ouverte} \\ c_{ij} + f_j & \text{sinon} \end{cases}$$

Si les contraintes de capacité sont activées, seules les installations avec une capacité suffisante sont considérées. Cette méthode garantit une solution initiale rapide mais qui peut dépendre grandement du premier client assigné en particulier si les coûts d'installation sont grands par rappport aux coûts d'assignation.

Cette méthode risque de tomber dans un minimum local, duquel on ne peut sortir qu'en réassignant plusieurs clients d'un seul coup (pour rentabiliser l'ouverture d'une nouvelle installation).

2. Heuristique moins gloutonne (LessGreedyInit) Cette approche introduit de la diversité tout en conservant une orientation vers des solutions de qualité. L'algorithme :

- Traite les clients par ordre décroissant de demande (clients les plus exigeants en premier, dans le cas avec capacités, sinon on ignore les demandes)
- Pour chaque client j, construit une liste de toutes les installations candidates avec leur coût total
- Trie cette liste par coût croissant
- Sélectionne aléatoirement parmi les 3 meilleures options (ou moins si moins de 3 candidats existent)

Cette méthode équilibre qualité et diversité en permettant des choix sous-optimaux contrôlés, ce qui peut aider à éviter les minima locaux lors du recuit simulé.

3. Heuristique aléatoire (RandomInit) Cette méthode maximise la diversité en générant des solutions complètement aléatoires. L'algorithme :

— Mélange aléatoirement l'ordre de traitement des clients
— Pour chaque client i, identifie toutes les installations capables de le servir (respectant les contraintes de capacité si activées)
— Sélectionne uniformément au hasard parmi toutes ces installations candidates

Cette approche garantit une grande diversité des solutions initiales, permettant au recuit simulé d'explorer des régions très différentes de l'espace des solutions.

Gestion des contraintes de capacité Toutes ces heuristiques intègrent les contraintes de capacité de manière paramétrique via le flag `capacity_constrained`. Quand ce flag est activé, chaque installation doit avoir une capacité suffisante pour servir les clients qui lui sont assignés. La capacité résiduelle est mise à jour dynamiquement lors de l'assignation des clients.

1.2 Structures de voisinage et perturbations du recuit associées

Représentation de la solution Une solution S est représentée par :

— Un vecteur binaire $\mathbf{x} \in \{0,1\}^m$ indiquant quelles installations sont ouvertes
— Un vecteur d'assignation $\mathbf{y} \in \{1,\dots,m\}^n$ spécifiant l'installation assignée à chaque client

Structures de voisinage Le voisinage est défini par trois types de perturbations possibles, chacune avec une taille de voisinage différente :

1. Perturbation d'ouverture (OpenFacilityMove) Ouvre une installation fermée j et réassigne les clients pour minimiser le coût total.

Taille du voisinage : $|\mathcal{N}_{open}| = m - |\mathcal{F}_o|$ où m est le nombre total d'installations et $|\mathcal{F}_o|$ le nombre d'installations actuellement ouvertes.

Le delta de coût est calculé comme :

$$\Delta = f_j + \sum_{i \in \mathcal{I}_j} (c_{ij} - c_{ij'})$$

où $\mathcal{I}_j$ est l'ensemble des clients réassignés à l'installation j, et j' était leur installation précédente.

Une idée, non implémentée ici, serait de trier les clients par $\frac{c_{ij}-c_{ij'}}{d_i}$ avant de réaliser l'assignation. Il s'agit d'une heuristique considérant que le remplissage de l'installation j est un sous-problème de sac à dos.

2. Perturbation de fermeture (CloseFacilityMove) Ferme une installation ouverte j et réassigne tous ses clients vers d'autres installations ouvertes selon un critère de coût minimal. S'il ne reste plus qu'une seule installation ouverte, ce move est considéré comme impossible.

Taille du voisinage : $|\mathcal{N}_{close}| = |\mathcal{F}_o|$.

$$\Delta = -f_j + \sum_{i \in \mathcal{I}_j} \min_{k \in \mathcal{F}_o \setminus \{j\}} c_{ik} - c_{ij}$$

où $\mathcal{F}_o$ est l'ensemble des installations ouvertes.

3. Perturbation d'échange (SwapFacilitiesMove) Ferme une installation ouverte j et ouvre une installation fermée k simultanément, optimisant les réassignations.

Taille du voisinage : $|\mathcal{N}_{swap}| = |\mathcal{F}_o| \times (m - |\mathcal{F}_o|)$ où chaque paire (installation ouverte, installation fermée) forme un move possible.

$$\Delta = f_k - f_j + \sum_{i \in \mathcal{C}} \min_{l \in \mathcal{F}_o \cup \{k\} \setminus \{j\}} c_{il} - c_{ij}$$

Gestion des contraintes de capacité Quand `capacity_constrained` est activé, chaque installation doit avoir une capacité suffisante pour servir les clients qui lui sont assignés. La réassignation des clients aux usines :

— ne concerne que les usines ouvertes/fermées lors de la perturbation

— se fait dans l'ordre dans lequel sont stockés les clients, ce qui implique que l'assignation n'est pas nécessairement optimale (des clients avec des coûts d'assignation faibles peuvent être refusés car la capacité de l'installation est déjà remplie).

Néanmoins, si une installation est ouverte puis refermée, des clients peuvent se retrouver assignés à une installation différente de leur affectation initiale.

1.3 Méthode de descente

Recuit simulé classique : L'algorithme utilise une décroissance exponentielle de la température :

$$T_{k+1} = \alpha \cdot T_k$$

avec $\alpha = 0.999$ et température initiale $T_0 = 10000$.

Critère d'acceptation : Un move est accepté avec probabilité :

$$P(\text{accept}) = \begin{cases} 1 & \text{si } \Delta \leq 0 \\ \exp\left(-\frac{\Delta}{T}\right) & \text{sinon} \end{cases}$$

Le recuit simple avec décroissance exponentielle donne de bons résultats mais s'arrête prématurément lorsque la température devient trop faible ($T < 1.0$ en partant de $T_0 = 10000$ avec un taux de refroidissement de 0.999) et qu'aucune amélioration n'est trouvée pendant 1000 itérations consécutives.

1.4 Metaheuristique proposée : recuit simulé à restart avec multithreading

Architecture parallèle : Le solveur lance plusieurs threads en parallèle, chacun avec une initialisation différente (gloutonne, moins gloutonne, aléatoire). Chaque thread exécute indépendamment le recuit simulé sur sa solution initiale.

Gestion des threads bloqués : Quand un thread atteint sa condition d'arrêt (température trop faible ou pas d'amélioration après un nombre d'itérations qu on détermine en fonction de la taille du voisinage), il se termine et redémarre, en repartant d'une solution créée avec le même mode d'initialisation. Le thread principal attend la fin de tous les threads via `thread::join()` et on s'arrête à la fin du temps imparti.

Chaque thread opère indépendamment et on ne partage pas d'information entre les threads, ce qui simplifie la gestion de la parallélisation.

Sélection de la meilleure solution : À la fin, le solveur compare les solutions finales de tous les threads et sélectionne celle avec le coût minimal :

1.5 Intégration des capacités

Les contraintes de capacité sont intégrées de manière paramétrique à tous les niveaux via le flag `capacity_constrained`. On constate qu'il est assez facile de respecter les capacités des les initialisations et de les maintenir au cours des moves, qui peuvent désormais fail et ne pas avoir de solution valide (notamment close).
L'implémentation n'est pas optimisée car on vérifie que la solution est valide mais suffisant pour converger en temps raisonnable.

2 Résultats

2.1 Tableau des résultats 20s d'exécution

Le tableau suivant présente les résultats obtenus par notre solveur multi-thread sur toutes les instances de test, en comparant les performances avec et sans contraintes de capacité, après 20 secondes d'exécution. Les valeurs sont exprimées en coût total de la meilleure solution trouvée.

Nous comparons dans la colonne **Gap relaxation** le résultat trouvé dans l'instance sans capacités avec la valeur optimale d'une relaxation continue du problème, cf. le calcul de cette valeur en annexe.

TABLE 1 – Résultats comparatifs avec et sans contraintes de capacité

Instance	Avec capacités	Sans capacités	Gap relaxation (%)	Gap sans capa (%)
cap71	932 615.75	932 615.75	10.4	0
cap72	977 799.40	977 799.40	15.1	0
cap73	1 010 641.45	1 010 641.45	18.4	0
cap74	1 034 976.98	1 034 976.98	20.4	0
cap101	797 508.73	796 648.44	20.8	0
cap102	854 704.20	854 704.20	28.7	0
cap103	893 782.11	893 782.11	33.7	0
cap104	928 941.75	928 941.75	37.7	0
cap131	793 439.56	793 439.56	25.7	0
cap132	851 495.33	851 495.33	33.8	0
cap133	893 076.71	893 076.71	39.3	0
cap134	928 941.75	928 941.75	43.3	0
capa	183 228 967.163	17 156 454.478	257.3	968
capb	77 123 122.536	12 979 071.581	253.8	494
capc	56 182 606.217	11 505 594.33	248.6	388

2.2 Forme des solutions optimales

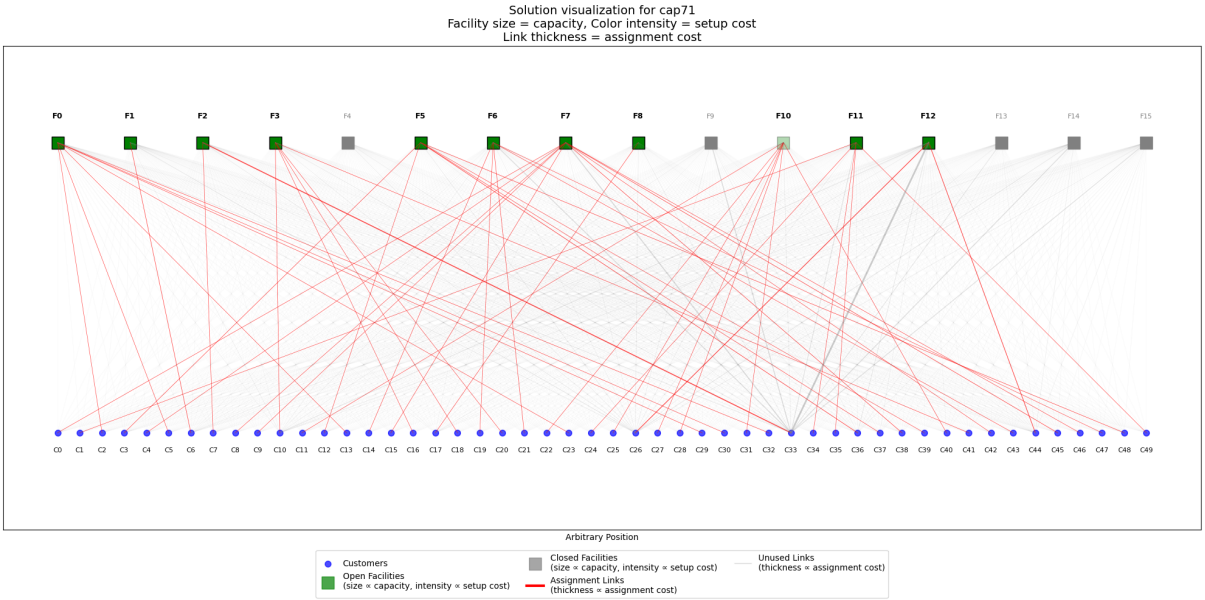

FIGURE 1 – Visualisation d'une solution optimale pour cap71 sans capacités

2.3 Statistiques des perturbations

Si on affiche les statistiques des perturbations, on peut voir que les perturbations de fermeture sont assez efficaces au début et que rapidement ce sont les échanges qui prennent le relais. On peut afficher un graphe avec les valeurs objectifs des 3 threads, les points colorés correspondant à une perturbation acceptée.

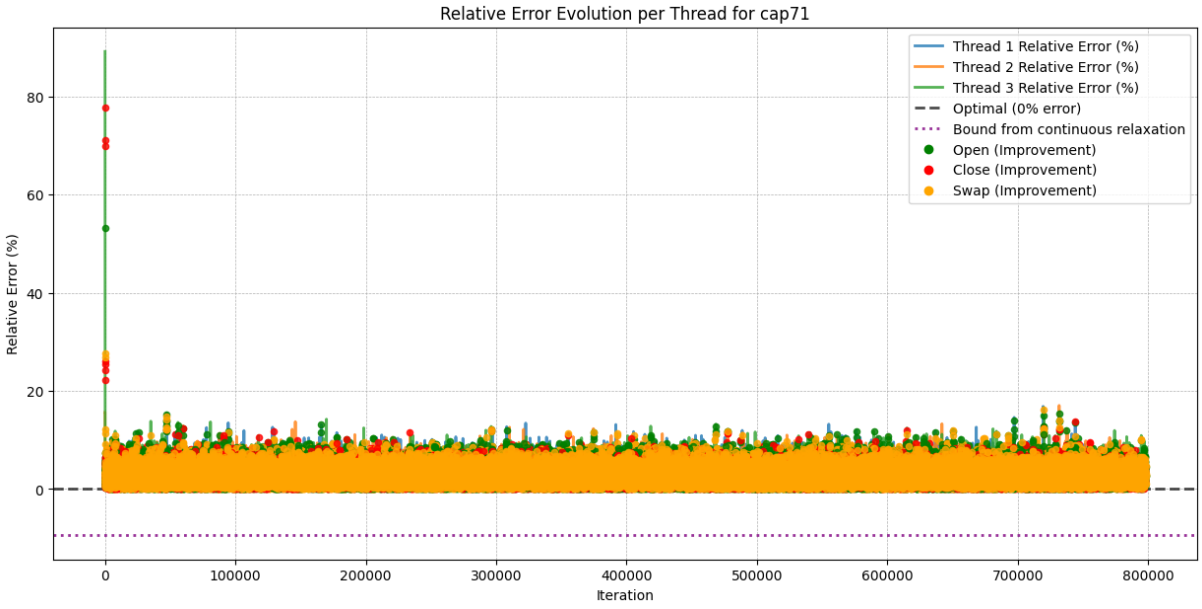

FIGURE 2 – Statistiques des moves pour cap71 sans capacités

Le «bruit »qu'on observe tout au long de l'exécution alors qu'on «devrait ⇝ *convergercorrespondau*

3 Annexes

3.1 Relaxation continue du problème de localisation discrète

On s'intéresse à une relaxation continue du problème étudié ici pour calculer facilement une borne inférieure de la valeur objectif optimale. Dans la modélisation du problème par un programme linéaire en variables 0-1, la contrainte $\forall i, j, y_{ij} \leq x_j$ est équivalente à la contrainte suivante : $\forall j, \sum_{i=1}^{N} y_{ij} \leq N x_j$.

Le programme associé à la relaxation continue de ce problème s'écrit alors :

$$\begin{cases} \min \quad f(x,y) & = \sum_{j=1}^{M} \left[\sum_{i=1}^{N} c_{ij} y_{ij} + f_j x_j \right] & \\ \text{s.c.} \quad \sum_{j=1}^{M} y_{ij} & = 1 & \forall i \\ \sum_{i=1}^{N} y_{ij} & \leq N x_j & \forall j \\ x_j & \in [0,1] & \forall j \\ y_{ij} & \in [0,1] & \forall i,j \end{cases}$$

On remarque qu'à l'optimum $\forall j, x_j = \frac{1}{N} \sum_{i=1}^{N} y_{ij}$, et $f(x,y) = \sum_{j=1}^{M} \sum_{i=1}^{N} \left(c_{ij} + \frac{f_j}{N} \right) y_{ij}$, d'où :

$$\min_{\forall i, \sum_{j=1}^{M} y_{ij}=1} f(x,y) = \sum_{i=1}^{N} \left[\min_{\sum_{j=1}^{M} y_{ij}=1} \sum_{j=1}^{M} \left(c_{ij} + \frac{f_j}{N} \right) y_{ij} \right] = \sum_{i=1}^{N} \min_{j} \left(c_{ij} + \frac{f_j}{N} \right)$$

.

En effet pour tout i, $\min_{\sum_{j=1}^{M} y_{ij}=1} \sum_{j=1}^{M} \left(c_{ij} + \frac{f_j}{N} \right) y_{ij} = \min A_i$ où A_i est l'ensemble des combinaisons convexes de $\left(c_{ij} + \frac{f_j}{N} \right)_j$