# Cinquante nuances de métaheuristiques pour le *job shop*

Julien Bernard[1]

Université Marie et Louis Pasteur, CNRS, institut FEMTO-ST, F- 25000 Besançon, France
julien.bernard@femto-st.fr

**Mots-clés** : *C++, Ordonnancement, Job Shop, Métaheuristiques*

## 1 Introduction

Les *templates* C++ permettent d'écrire du code générique qui peut être instancié de multiple fois. Dans cet article, nous montrons l'utilisation des *templates* ainsi que des *concepts* en C++ pour l'implémentation d'une bibliothèque destinée à la résolution de problèmes d'ordonnancement de type *job shop* à l'aide de métaheuristiques.

L'origine de cette bibliothèque provient de l'étude des codages de solutions pour le problème du *job shop* (JSP) ainsi que l'étude des opérateurs de voisinage associés à ces codages dans l'implémentation d'une métaheuristique, la recherche tabou [2]. Ces travaux ont été prolongés à une autre métaheuristique, un algorithme génétique [1], et à une variante plus complexe du problème, le *job shop* flexible avec transport (JSPT).

Dans l'implémentation de ces deux métaheuristiques, il est apparu un premier ensemble de notions réutilisables formant un paramétrage externe de la métaheuristique qui seront abordés dans la section 2. Puis pour chacune de ces métaheuristiques, en particulier l'algorithme génétique, un second ensemble de paramètres internes s'est dégagé dont nous discuterons dans la section 3. L'utilisation de C++ s'est imposée d'une part par les fonctionnalités de programmation générique et d'autre part par sa rapidité d'exécution. Le but premier était de ne pas avoir à réimplémenter des dizaines de variantes pour chacun des algorithmes de métaheuristiques.

## 2 Concepts pour l'ordonnancement de *job shop*

Les *concepts* C++ sont apparus dans la version C++ 20. Ils permettent de contraindre des arguments de *template* et donc de détecter au plus tôt dans la compilation les erreurs d'instanciation, en plus de donner des messages d'erreur plus explicites. Dans notre bibliothèque, nous avons identifié un certain nombre de *concepts*.

Tout d'abord, nous avons identifié le *concept* `Instance` qui représente une instance de problème d'ordonnancement et qui basiquement permet d'obtenir le nombre de tâches et le nombre de machines du problème. Nous avons raffiné ce *concept* en `ShopInstance` pour le JSP et le JSP flexible en ajoutant des informations sur les gammes opératoires ; puis en `ShopTransportInstance` pour les variantes avec transport en prenant en compte les ressources de transport et les stations de chargement/déchargement.

Ensuite, une solution au problème représentée à l'aide d'un codage a été modélisée avec les *concepts* `Input` et `InputArray` associés à un ensemble d'opérations qui permettent leur manipulation par divers opérateurs utilisés dans les métaheuristiques (voisinage, mutation, croisement). De manière orthogonale, les opérateurs de voisinages ont été regroupés sous le concept de `Neighborhood` qui permet de créer un ou plusieurs voisins à partir d'une solution.

Enfin, nous avons défini le *concept* `Engine`, c'est-à-dire un algorithme qui, à partir d'une solution et d'une instance (possiblement générique), est capable de calculer (ou non) un ordonnancement valide pour le problème. Les classes qui implémentent ce *concept* sont souvent capables de traiter les versions flexibles aussi bien que non-flexibles d'une même variante de

*job shop*, encore une fois grâce à la programmation générique. Ces moteurs sont eux-même parfois paramétrés par des fonctions de choix parmi des ensembles d'opérations candidates.

L'ensemble de ces concepts permettent de décrire les paramètres externes aux algorithmes de métaheuristiques, c'est-à-dire ceux qui peuvent être réutilisés indépendamment de la métaheuristique choisie.

## 3 Métaheuristiques génériques

Les métaheuristiques sont par nature des algorithmes génériques. Il est donc naturel de les programmer de manière générique, en faisant abstraction autant que possible des éléments concrets manipulés. Nous avons ajouté deux *concepts* propres à toutes les métaheuristiques. Le premier est le *concept* `Criterion` qui permet, à partir d'une instance et d'un ordonnancement, de calculer une métrique à optimiser, et de pouvoir comparer deux métriques pour savoir laquelle est la meilleure. Le second est le *concept* `Termination` qui permet de déterminer la condition d'arrêt de l'exécution de la métaheuristique.

La recherche tabou a très peu de paramètres en dehors de la longueur de la liste tabou. Son implémentation générique, à partir de tous les concepts précédemment définie, a donc été assez directe. L'algorithme génétique a nécessité d'ajouter de la généricité au niveau de l'algorithme de sélection, de l'algorithme de mutation (qu'on a identifié à un algorithme de voisinage) et de l'algorithme de croisement. Au final, ces deux algorithmes pourraient encore sans doute bénéficier de plus de généricité mais comme ce n'était pas l'objet de notre étude, ces versions basiques ont été suffisantes.

Tout ce travail a permis d'instancier 15 versions de recherche tabou dans le cadre de [2], étendues à 32 actuellement, et 15 versions d'algorithme génétique dans le cadre de [1]. Nous travaillons actuellement sur le problème du FJSPT. Ce problème permet d'être attaqué de multiples manières : ordonnancement des opérations, affectation des opérations aux machines, affectations des tâches de transport aux ressources de transport. Il existe donc de multiples familles de codage pour résoudre ce problème, ainsi que d'opérateurs de voisinage spécifiques. Au final, à l'heure actuelle, nous sommes capable de générer 670 variantes de recherche tabou pour le problème du FJSPT.

## 4 Conclusion

La programmation générique à l'aide de *template* et de *concepts* en C++ a permis d'implémenter de nombreuses variantes d'algorithmes métaheuristiques sans avoir à dupliquer les efforts pour chaque cas. Ce travail est toujours en cours pour permettre de créer toujours plus de variantes sur d'autres problèmes d'ordonnancement et d'autres métaheuristiques. La bibliothèque, baptisée `libsched`, est disponible sur un dépôt `git` [1] avec une licence libre.

## References


[1] Karla Breschi, Julien Bernard, Hervé Manier, and Marie-Ange Manier. Efficiency variations of solution encodings and neighborhood operators with different metaheuristics for the job-shop scheduling problem. In Hamid R. Arabnia, Leonidas Deligiannidis, Farzan Shenavarmasouleh, Soheyla Amirian, and Farid Ghareh Mohammadi, editors, *Computational Science and Computational Intelligence*, pages 220–234, Cham, 2025. Springer Nature Switzerland.

[2] Israël Tsogbetse, Julien Bernard, Hervé Manier, and Marie-Ange Manier. Influence of encoding and neighborhood in landscape analysis and tabu search performance for job shop scheduling problem. *European Journal of Operational Research*, 319(3):739–746, 2024.


1. `https://github.com/SchedulingLab/libsched`