INF6102 - Métaheuristiques appliquées au génie informatique

Devoir 3 - Optimisation de la production de meubles

Guillaume Thibault
g.thibault@polymtl.ca

Guillaume Blanché
guillaume.blanche@polymtl.ca

11 avril 2023

1 Description du problème

L'atelier LaTulipe, un fabricant de meubles sur mesure, fournit des meubles à de grands clients institutionnels tels que des hôpitaux ou des universités. Chaque commande de meubles est divisée en plusieurs tâches qui sont définies par des relations de précédence, des durées et nécessitent une certaine quantité de ressources. Les relations de précédence expriment les tâches qui ne peuvent commencer avant qu'une tâche en question ne soit terminée. Le but ultime est de minimiser le temps total nécessaire pour terminer toutes les tâches. Dans cet ensemble de tâches, chaque tâche a un temps de traitement et chaque ressource renouvelable a une capacité, et il peut y avoir des relations de précédence entre les tâches.

Ce probème peut être mathématique simplifié sous la forme suivante ; Le but est de minimiser le makespan tout en respectant les contraintes de précédences (2) et de capacité des ressources (3). De plus, une dernière Contrainte (4) indique que les activités ne peuvent pas commencer avant le temps zéro, c'est-à-dire le début de la planification.

2 Méthode de résolution

2.1 Recherche à voisinage variable

La première implémentation réalisée dans le cadre de ce travail consiste en un algorithme de construction par recherche à voisinage variable. Cette méthode de construction de solutions se base sur la priorisation des tâches les plus critiques, c'est-à-dire celles qui ont le plus grand nombre de tâches en attente qui en dépendent. Lorsqu'une tâche, identifiée comme étant la tâche tâche_j, est sélectionnée, toutes les tâches réalisées avant ou en même temps que celle-ci sont préservées et copiées dans la nouvelle solution en cours de construction, tandis que toutes les tâches provenant après celle-ci ne sont pas considérées. Ensuite, la tâche tâche_j est insérée le plus tôt possible dans la nouvelle solution en cours de construction. Une fois insérée, le reste de la solution est construit de manière semi-aléatoire. Cette construction est effectuée en sélectionnant les tâches à insérer de manière aléatoire en échantillonnant une distribution construite en temps réel qui pondère le nombre de tâches débloquées par chaque tâche ($+1$, pour ne pas exclure les tâches qui ne débloquent aucune autre).

Cette méthode permet à l'algorithme de détruire une solution existante, de se concentrer sur l'optimisation d'un voisinage spécifique, puis de reconstruire le reste de la solution de manière semi-aléatoire, permettant d'équilibrer l'exploration et l'exploitation. Cette approche de construction de solutions est très rapide, ce qui permet de générer un grand nombre de solutions dans le temps imparti, permettant ainsi de trouver ou de se rapprocher très près de la solution optimale. En d'autres termes, cette méthode utilise une stratégie de métaheuristique de recherche par voisinage pour construire une solution en se concentrant sur les tâches les plus critiques, ce qui se traduit par une accélération significative de la construction de solutions.

Algorithm 1 Algorithme de recherche par voisinage variable

$solution_{best} \leftarrow null$
while CPU time left $\neq 0$ **do**
  $solution \leftarrow$ generateRandomValidSolution()
  $neighborhoods\_to\_priorize \leftarrow$ getSortedPrioritizedNeighborhood()
  $k \leftarrow 0$
  $k_{max} = NUMBER\_OF\_JOBS$
  while $k \neq k_{max}$ **do**
    $solution_{new} \leftarrow$ localSearch($solution, k, neighborhoods\_to\_priorize$)
    $solution, k \leftarrow$ neighborhoodChange($solution, solution_{new}, k$)
  end while
  $solution_{best} \leftarrow$ getBestSolution($solution, solution_{best}$)
end while
return $solution_{best}$

Algorithm 2 Algorithme neighborhoodChange($solution$, $solution_{new}$, k)

$score_{solution} \leftarrow$ getScore($solution$)
$score_{solution_{new}} \leftarrow$ getScore($solution_{new}$)
if $score_{solution_{new}} \leq score_{solution}$ **then**
  return $solution_{new}$, k
else
  return $solution$, $k + 1$
end if

Algorithm 3 Algorithme localSearch($solution$, k, $neighborhoods\_to\_priorize$)

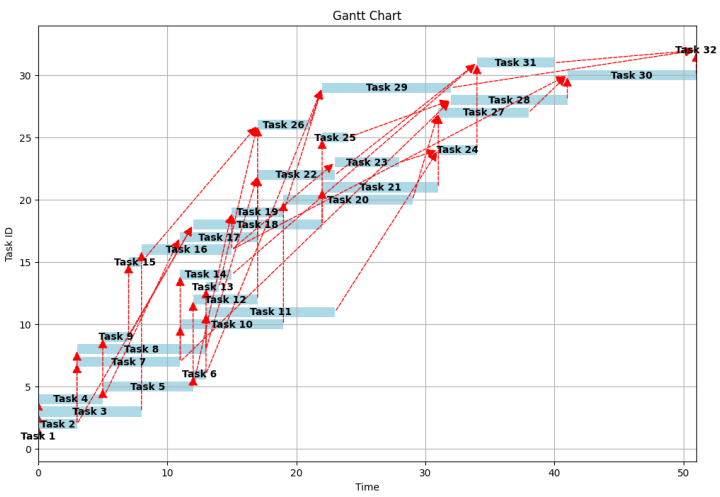
$solution_{new} \leftarrow$ generateEmptySolution()
$job_k \leftarrow neighborhoods\_to\_priorize[k]$
$solution_{new} \leftarrow solution_{new} +$ jobsBeforeJobK($solution$, job_k)
$available\_ressources\_through\_time \leftarrow$ getRessourcesAvailableThroughtTime($solution_{new}$)
$original\_timestep_k \leftarrow solution[job_k]$
$timestep \leftarrow 0$
while $timestep \leq original\_timestep_k$ **do**
  if predecessors(job_k) $\in$ jobDoneAt($solution_{new}$, $timestep$) **then**
    $ressources\_needed\_until = timestep +$ duration(job_k)
    $ressources\_in\_interval \leftarrow ressources\_through\_time[timestep : ressource\_needed\_until]$
    $job_k\_ressources \leftarrow$ ressourceNeededToDoJob(job_k)
    if min($ressources\_in\_interval - job_k\_ressources$) ≥ 0 **then**
      $solution_{new}[job_k] = timestep$
      Break
    end if
  end if
  $timestep \leftarrow timestep + 1$
end while
$solution_{new} \leftarrow$ randomdlyInsertMissingJob($solution_{new}$) ▷ Job only added at the end, after job_k
$solution_{new} \leftarrow$ removeDelay($solution_{new}$) ▷ Find and remove wasted time between jobs
return $solution_{new}$

3 Résultats

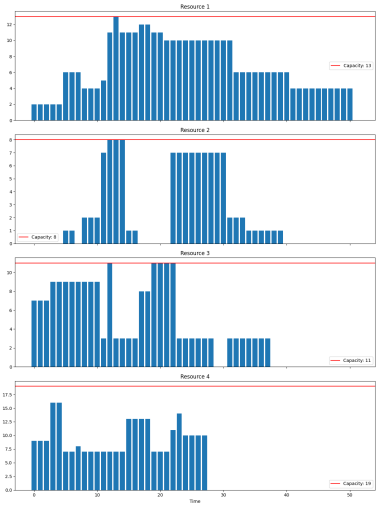
3.1 Résultats sur les instances fournies

Instance	Makespan	Optimum
A_30	51	51
B_30	43	43
C_60	80	74
D_60	122	114
E_90	96	78

TABLE 1 – Makespan trouvé sur les instances fournies

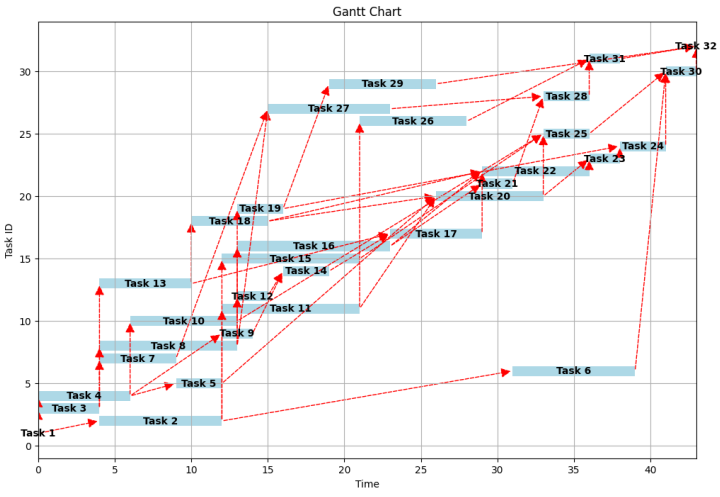


(a) Le diagramme de Gantt illustrant le planning des tâches

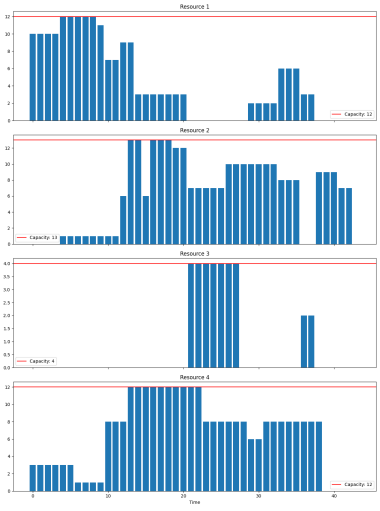


(b) Disponibilité des ressources au travers du temps

FIGURE 1 – Instance A_30

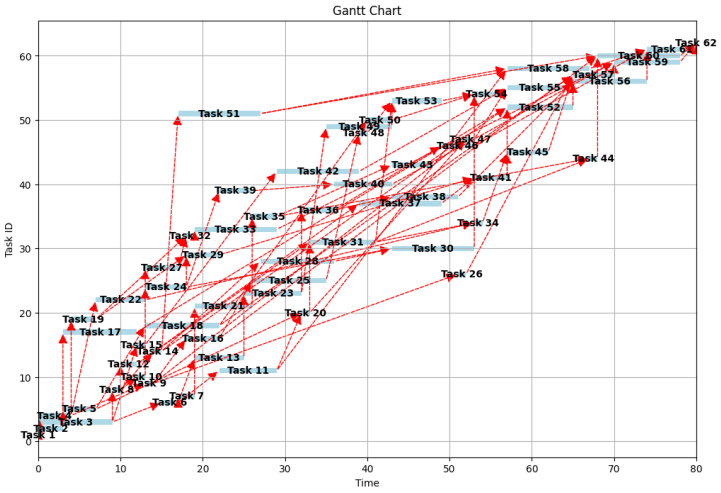


(a) Le diagramme de Gantt illustrant le planning des tâches

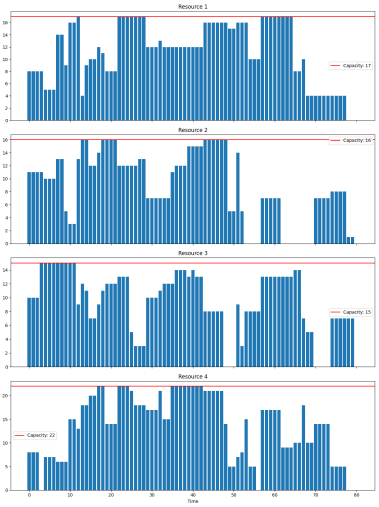


(b) Disponibilité des ressources au travers du temps

FIGURE 2 – Instance B_30

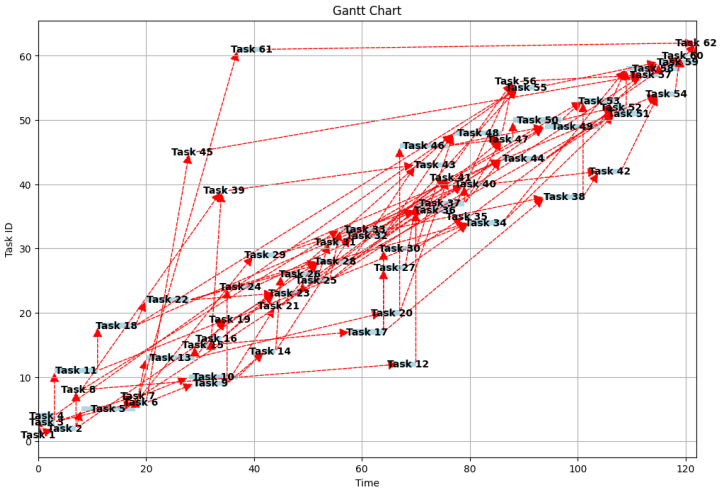


(a) Le diagramme de Gantt illustrant le planning des tâches

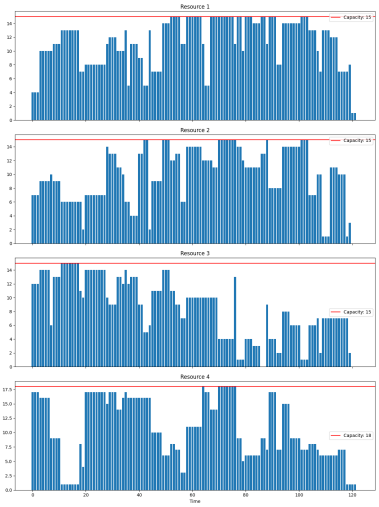


(b) Disponibilité des ressources au travers du temps

FIGURE 3 – Instance C_60

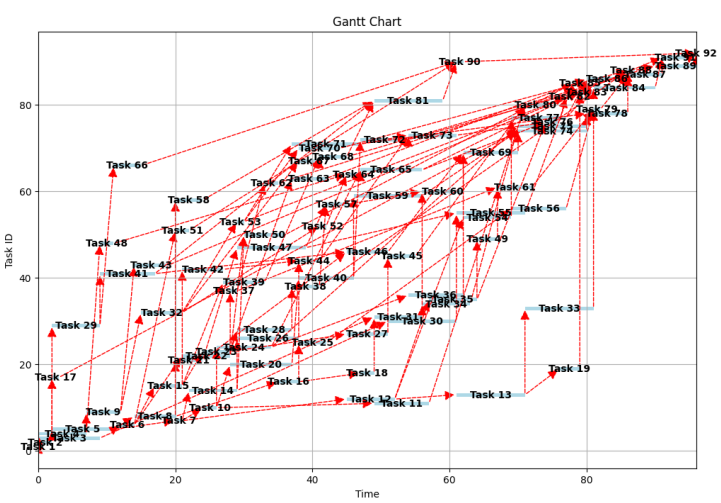


(a) Le diagramme de Gantt illustrant le planning des tâches

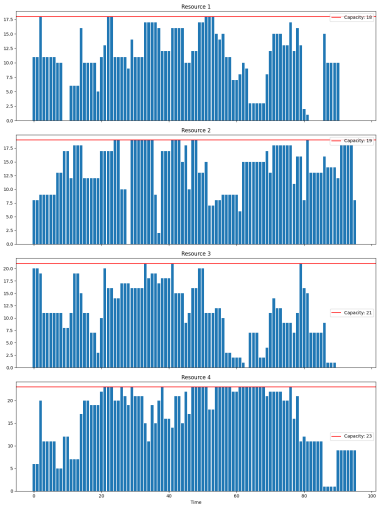


(b) Disponibilité des ressources au travers du temps

FIGURE 4 – Instance D_60



(a) Le diagramme de Gantt illustrant le planning des tâches



(b) Disponibilité des ressources au travers du temps

FIGURE 5 – Instance E_90

3.2 Analyse

En analysant les résultats obtenus, nous constatons que l'algorithme que nous avons développé est capable de trouver les solutions optimales pour les instances de petite taille. Cependant, lorsque le nombre de tâches augmente, il devient de plus en plus difficile pour l'algorithme de trouver des solutions optimales. Une observation que nous pouvons faire est que notre méthode de résolution n'applique pas efficacement des pénalités pour les délais très longs entre les tâches, ce qui peut conduire à l'exécution de certaines tâches qui ne contribuent pas directement à l'achèvement des tâches subséquentes.

Ce problème est dû à l'exécution de la recherche locale, qui tente d'exécuter les tâches le plus tôt possible sans prendre en compte le fait que cela pourrait bloquer l'exécution d'autres tâches plus prioritaires à court terme. Cependant, il est plus complexe de tenir compte de cette priorité lors de la construction des priorisations de voisinage. Une solution envisageable serait d'intégrer une méthode d'apprentissage en temps réel capable d'apprendre à partir d'une solution optimisée en détectant les tâches qui sont exécutées trop rapidement.

Cette méthode permettrait de détecter les tâches qui sont exécutées de manière inefficace et d'adapter en conséquence la résolution du problème. Cela pourrait également conduire à l'amélioration des performances de l'algorithme pour les instances de grande taille et contribuer à la résolution de problèmes complexes dans divers domaines.

4 Conclusion

En somme, nous avons conçu un algorithme novateur qui intègre une recherche par voisinage variable avec une stratégie de destruction, d'optimisation locale et de reconstruction, le tout en respectant les contraintes définies au préalable. Bien que nos résultats ne soient pas optimaux pour les instances les plus complexes, notre algorithme parvient à ordonner rapidement les tâches avec une bonne qualité et sans dépasser les ressources allouées ou les délais impartis. Nous avons également identifié une piste d'amélioration pour l'avenir en ajoutant une méthode d'apprentissage pour mieux déterminer les tâches à prioriser lors de la recherche par voisinage variable. Cette méthode devra s'appuyer sur les solutions générées précédemment pour affiner la sélection des tâches prioritaires.