INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Méthodes approchées pour la résolution de problèmes de type “job shop”

Auteur :
M. Pierre-Arnaud Auque

Version 1.0 du
19 mars 2020

Table des matières

1 Introduction

Le problème de jobshop est connu, et le but est d'ordonnancer un ensemble de "job" sur un ensemble de machine. Chaque "job" est sequencé avec un ensemble de tâche, et ces tâches sont associées à des machines spécifiques. Le "makespan", c'est à dire le temps total pour que toutes les activités soient terminées, est la métrique qui sera utilisée pour comparer les différentes solutions d'ordonancements. Le but est de minimiser ce "makespan", en un minimum de temps / itération. Ainsi nous allons voir plusieurs heuristiques, qui permettent de se rapprocher de l'ordonancement optimal.

— Une méthode exacte via la bibliothèque docplex.
— Dans un second temps un algorithme glouton.
— Dans un troisème temps un algorithme de recherche locale.
— Pour finir par un algorithme génétique.

En dehors de la méthode exacte, la représentation par répétition (ou Vecteur de Bierwirth) a été choisie pour représenter un ordonancement donné des activités. Ce choix permet de s'abstraire des contraintes d'ordonancement, à la fois au niveau des tâches et des machines. Une solution représentée par ce vecteur est toujours réalisable. Il est donc plus facile, à partir de cette représentation d'effectuer des permutations (par exemple pour la recherche de voisinages).

2 Méthode exacte

La méthode exacte se sert du package de docplex afin de trouver des solutions aux instances. Comme on peut voir sur la table 1, le package est trés performant et trouve rapidement l'optimal des instances.

nom	MakeSpan	temps (seconde)	Optimum
ft20	55	<0.01	55
ft10	930	1.71	930
ft20	1165	0.32	1165
la01	666	<0.01	666
la10	958	0.01	958
ta01	1231	8.61	1231

TABLE 1 – Makespan et temps de la méthode exacte (docplex).

3 Algorithme Glouton

3.1 Heuristique générale

L'algorithme glouton consiste à allouer des tâches aux différents machines, en fonction d'une règle de priorité. Les règles implémentérs dans le cadre de ce projet sont "Durée la plus courte" (SPT) et "Durée la plus longue" (LPT).

Pour chaque pas de temps t, l'algorithme vérifie si une resource est libre et si une tâche est terminée. Si c'est deux conditions sont vérifiés, on prendra dans la liste des tâches faisables, celle qui est la plus prioritaire (selon STP ou LPT). Les pseudo-codes sont disponibles dans l'annexe C.

nom	SPT	LPT	Optimum
ft06	90	79	55
ft10	1075	1299	930
ft20	1268	1633	1165

TABLE 2 – Makespan de l'algorithme glouton en fonction de différentes priorités (instances de type 'ft').

3.2 Heuristique randomisée

L'Heuristique randomisée se base sur l'heuristique générale, en ajoutant une part d'aléatoire dans le choix des tâches à allouée. Dans la table 3, on peut voir le résultats moyens (100 tirages) sur différentes instances avec des taux de randomisation différents (1 est égal à une heuristique entièrement aléatoire). En moyenne, l'algorithme randomisée est moins bon que l'heuristique générale, surtout pour des problèmes de grand taille, où l'espace des solutions est trés grands.

nom	SPT	LPT	Optimum
ft06, rnd= 0 (non randomisé)	90.0	79.0	55
ft06, rnd= 0.2	80.23	74.07	55
ft06, rnd= 0.5	71.85	69.89	55
ft06, rnd= 1	68.56	69.53	55
ft10, rnd= 0 (non randomisé)	1075.0	1299.0	930
ft10, rnd= 0.2	1150.35	1286.14	930
ft10, rnd= 0.5	1199.6	1275.38	930
ft10, rnd= 1	1227.35	1240.99	930
ft20, rnd= 0 (non randomisé)	1268.0	1633.0	1165
ft20, rnd= 0.2	1339.6	1598.89	1165
ft20, rnd= 0.5	1390.74	1581.68	1165
ft20, rnd= 1	1528.92	1527.58	1165

TABLE 3 – Algorithme glouton sur instances de type 'ft', rnd = 1 correspond à une heuristique entièrement randomisée.

4 Recherche locale

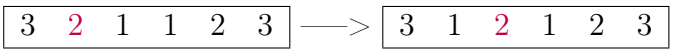
La recherche locale, consiste à explorer les voisinages d'une solution donnée. Les métriques d'évalutions sont le makespan et la durée de recherche du voisinage (ou nombre d'itération). Les paramètres de cette recherche locale sont :

— la profondeur de la recherche,
— le nombre de voisins à explorer,
— un critère de stagnation.

L'algorithme va estimer parmis tout les voisins possibles d'une solution, le makespan optimal, jusqu'a ce que au moins un critère soit atteint (profondeur ou stagnation (i.e pas de nouvel optimal depuis x itérations)). La recherche de voisinage est faite via des permutations, des décalages, et des inversions.

4.1 Permutation

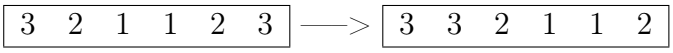
Des permutations deux à deux sur le vecteur de répétition, sont un moyen simple d'explorer le voisinage d'une solution. Cependant la taille du voisinage est de l'ordre de n^2, qui est le nombre de permutation possible.



Le code ci-dessous permet à partir d'une liste, de générer l'ensemble des permutations deux à deux possible :

4.2 Décalage

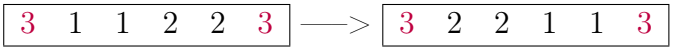
Le décalage, permet à partir d'un indice, de décaler l'ordonancement vers la droite. Une liste de taille n possède n décallage possible.



Le code ci-dessous permet à partir d'une liste, de générer l'ensemble des décalages possibles :

4.3 Inversion

L'inversion, permet à partir de deux indices, d'inverser l'ordonancement entre ces deux indices. Une liste de taille n possède $\frac{n(n-1)}{2}$ inversion possible.



4.4 Analyse des résultats

L'ensemble des méthodes de voisinage ont été testé sur différentes instances (ft06,ft10, ft20, la01, la20, ta10). En partant de l'heuristique gloutonne de type SPT. Les résultats sont rassemblés dans le tableau 4. Comme anticipé au vu des nombreux voisinages, leurs explorations totales prends du temps. Néanmoins sur les instances complexes, il est impossible de faire à peine mieux que l'algorithme glouton. La recheche locale dépends du point de départ, et donc il se peut que à a partir de STP ou LPT, la recheche de solution reste bloquée dans un minimum local.

nom	makespan (voisinage)	temps (s)	itérations	Glouton	Optimum
ft06	55	87.99	109	90	55
ft10	997	792.06	56	1075	930
ft20	1267	392	25	1268	1165
la01	697	89	42	734	666

TABLE 4 – Résultats de la recherche de voisinage sur les instances de types ft.

5 Algorithme génétique

La métaheuristique choisie, afin d'optimiser la recherche de solution et surtout afin de sortie des minima locaux, est l'algorithme génétique. Pour cela différentes fonctions reproduisant au mieux le fonctionnement génétique ont été dévéloppées. Les hyperparamètres de chaque fonction, ont été testé, ceux présentés dans ce rapport semble les plus performants. Néanmoins, il semble qu'ils doivent être adaptés à chaque problème.

5.1 Gènése

La génése est la création de la population initiale. Cette population initiale contient les solutions relatives au algorithme SPT et LPT, ainsi qu'un pourcentage de leur voisinage (via permutation), complété par des solutions aléatoires (fig. 1). La taille de la population initiale est un hyperparamètre important.

5.2 Evaluation

La population générée est ensuite évaluée, la métrique d'évaluation (=fitness) est l'inverse du makespan. Donc plus le makespan est petit, plus la fitness sera grande. Une probabilité de sélection est affectée à chaque individu basée sur leur rang vis à vis de leur fitness. Plus le fitness sera grand plus la probabilité de sélection sera grande, tel que :

$$P_i = \frac{rang_i}{n(n+1)/2}.$$

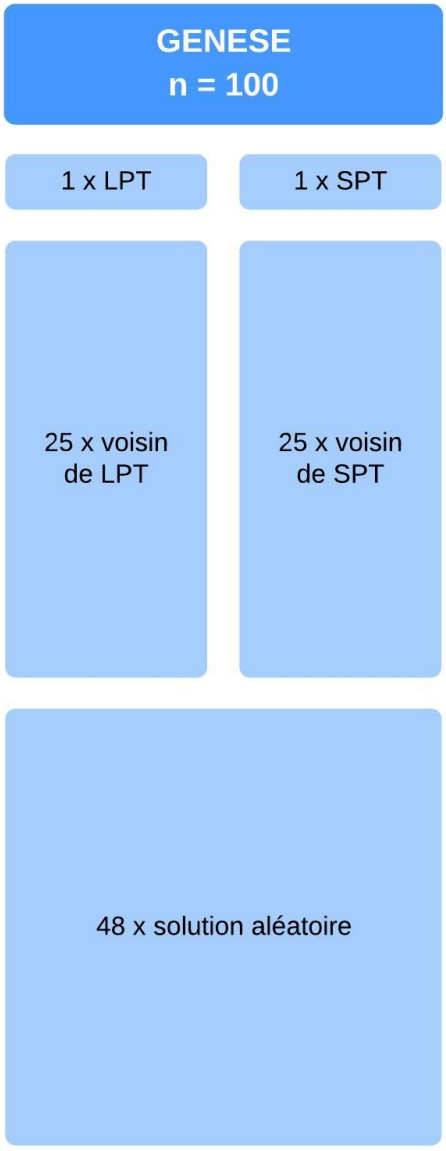


FIGURE 1 – Répartition de la population initiale lors de la gènese.

5.3 Croisement

Le croisement d'individus va permettre d'obtenir la génération suivante. Le module de croisement peut être configuré pour que :

— les parents soient sélectionnés par duel (tirage aléatoire sans remise de deux individus, et selection du meilleur).

— les parents soient sélectionnés par roulette (selon les probabilités définies lors de l'évaluation).

Dans le deuxième cas, un "bon" individu à plus de chance d'être croisé qu'un "mauvais" individus, car les probabilités de tirage sont fonctions du makespan de la solution/individu. Le croisement à un probabilité $\alpha = 0.85$ de réussir (fig. 2). Chaque croisement genère deux enfants différents (l'un qui va plus hérité de la mère et l'autre plus du père).

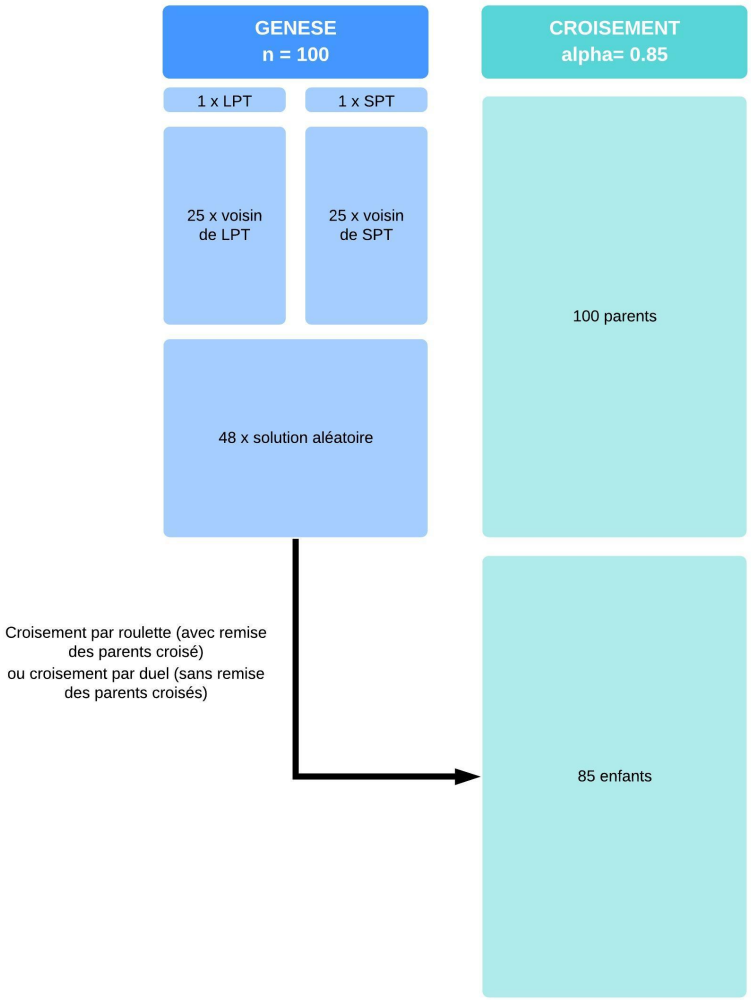


FIGURE 2 – Répartition de la population lors de la croisement.

L'opérateur de croisement va définir comment les gènes du père et de la mère vont être mélangés. Trois opérateurs de croisement ont été choisi afin de pouvoir explorer un maximum l'espace des solutions :

— Croisement en 1 point, consiste à choisir un indice, ce qui est à gauche de l'indice est héritée d'un parent, ce qui est à droite de l'indice est complété par l'autre parent (completion des job/taches manquantes).

père	3 ↓	2 ↓	1 ↓	1	2	3
enf1.	3	2	1	3	1	2
mère	3	3 ↑	2	1	1 ↑	2 ↑

père	3	2	1 ↓	1 ↓	2 ↓	3
enf2.	3	3	2	1	1	2
mère	3 ↑	3 ↑	2 ↑	1	1	2

— Croisement en 2 points, consiste à choisir deux indices, ce qui hors des deux indices est hérité par un parent, ce qui est entre les indices est complété par un parent (completion des job/taches manquantes)

père	3 ↓	2 ↓	1	1	2	3 ↓
enf1.	3	2	1	1	2	3
mère	3	3	2	1 ↑	1 ↑	2 ↑

père	3	2	1 ↓	1 ↓	2 ↓	3
enf2.	3	3	1	1	2	2
mère	3 ↑	3 ↑	2	1	1	2 ↑

— Croisement des jobs, les gènes vont être définis par les jobs qui viennent soit du père, soit de la mère (et non un mélange de deux). Sur chaque job, un tirage binaire va etre effectué pour déterminer, de quel parent sera issu le job (par exemple si $P_1 = 1$, le job 1 aura le même ordre que le père dans le vecteur à répétition). Dans l'exemple ci-dessous, l'enfant héritera du père de la séquence du job 1 et 2, et de la séquence du job 3 de la mère.

Job 1	Job 2	Job 3
1	1	0

père	3	2 ↓	1 ↓	1 ↓	2 ↓	3
enf.	3	2	1	1	2	3
mère	3 ↑	3 ↑	2	1	1	2

5.4 Mutation

Aprés le processus de croisement, les enfants engendrés, ont une chance de muter, cette mutation permet de sortir des minima locaux, et ajoute encore un part d'aléatoire pour potentiellement accelerer la convergence. La mutation se produit avec une probablité de (par opérateur de mutation) :

$$\beta = 0.15(1 + f)$$

,où f est un facteur multiplicatif, $f = \frac{nb_{generationsschgt}}{10}$. Ce facteur augmente des lors qu'aucun nouvel optimum a été trouvé à chaque génération. Ainsi un individu à plus de chance de muter s'il n'y à pas eu de nouvel optimum au génération précédente. Les chances de mutations doublent toute les 10 générations sans changement (sans nouvel optimal).

On distingue quatres opérateurs de mutation :

— Une permutation deux a deux dans le vecteur a répétition.

— Une inversion, un décalage, une insertion (cf voisinage).

Les quatres opérateurs de mutations peuvent se produire sur un même enfant (fig. 3).

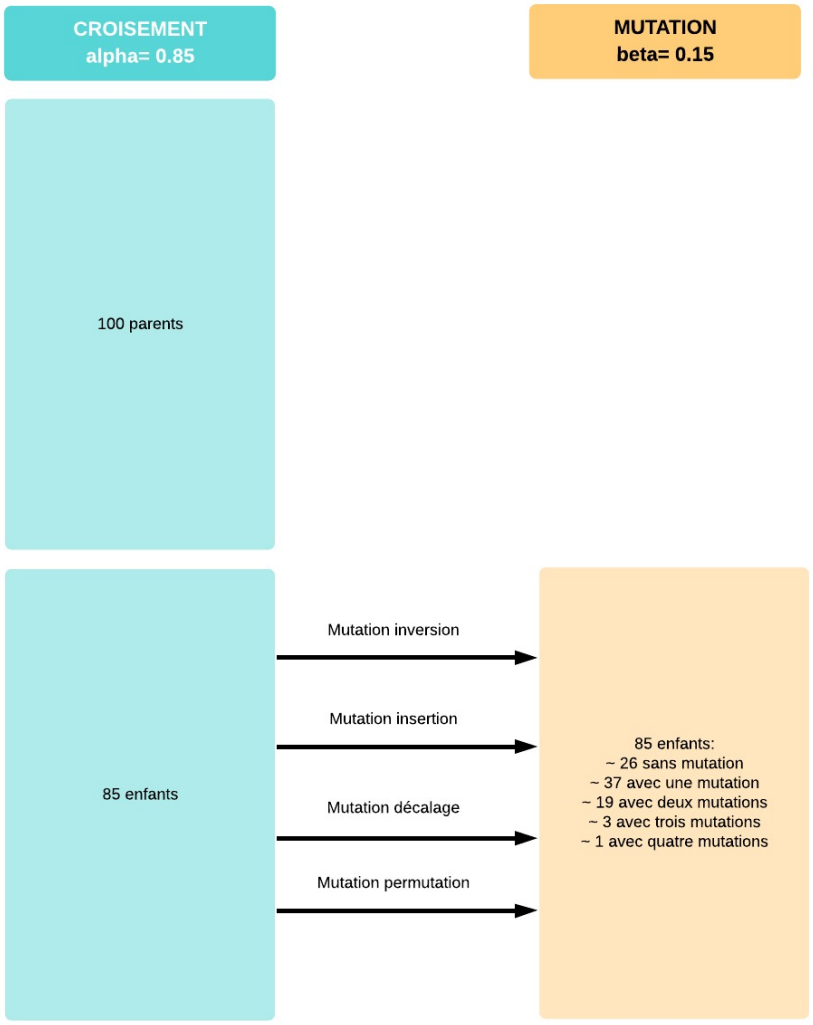


FIGURE 3 – Répartition de la population lors de la mutation.

5.5 Sélection

L'étape de sélection peut soit (configurable) :

— choisir aléatoirement (selon les probabilités calculée par rapport à la fitness) un nombre d'individu donné, afin de ne pas perdre les meilleurs individus. L'étape de sélection va aussi ajouter 10% (de la taille de la population) d'individus supplémentaires. Ces individus supplémentaires sont tous les mêmes et correspondent à la meilleur solution trouvée jusqu'a présent. Ceci permet de conserver les meilleurs solutions et permet à l'algorithme de converger plus rapidement. Il est aussi possible de rajouter des individus aléatoires à la population, mais cela n'a pas été retenu, car peu concluant sur les résultats.

— choisir les meilleurs individus parmis les enfants et les parents.

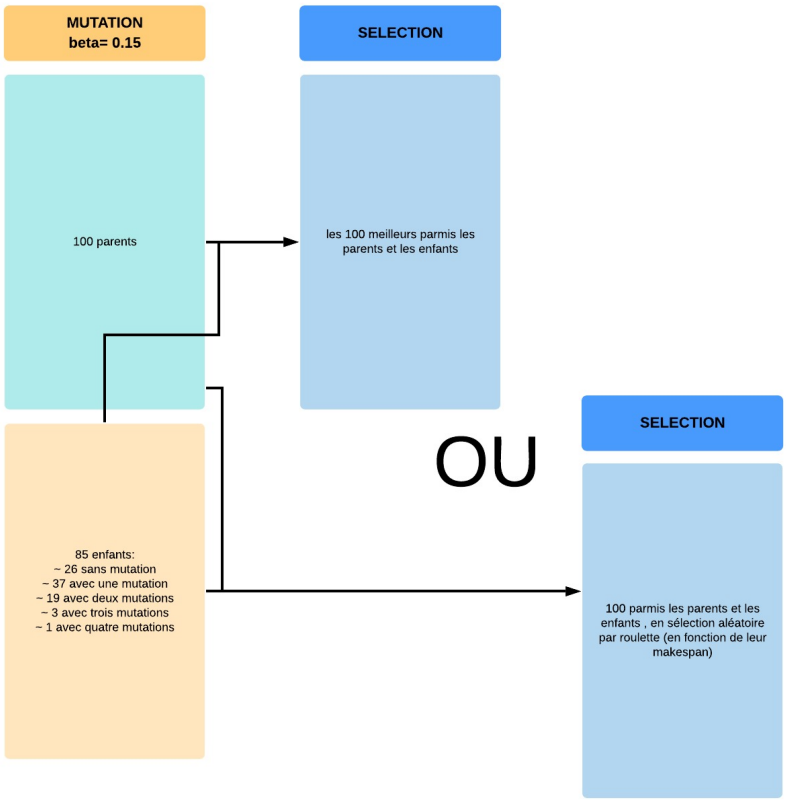


FIGURE 4 – Répartition de la population lors de la selection (deux manières différentes).

5.6 Résultats

L'algorithme génétique a été testé sur les instances ft06, ft10 ,la01. Les résultats sur les autres instances sont peu concluants. On peut voir que sur la table 5, que sur les petites instances (ft06 et la01) l'algorithme converge assez souvent et rapidement vers la solution optimale. Notament plus rapidement que sur la recherche de voisinage classique. Beaucoup de paramètre ainsi que de combinaison de méthode reste à tester afin d'avoir un algorithme robuste pour plusieurs type d'instance. De plus le temps de calcul peut être important afin de trouver une solution convenable et sortir des mimima locaux. Le tracé des courbes de makespan en fonction des générations permet d'apporter une aide à la décision sur les paramètres à choisir (convergence trop rapide, etc...), un exemple est donné sur la figure 5.

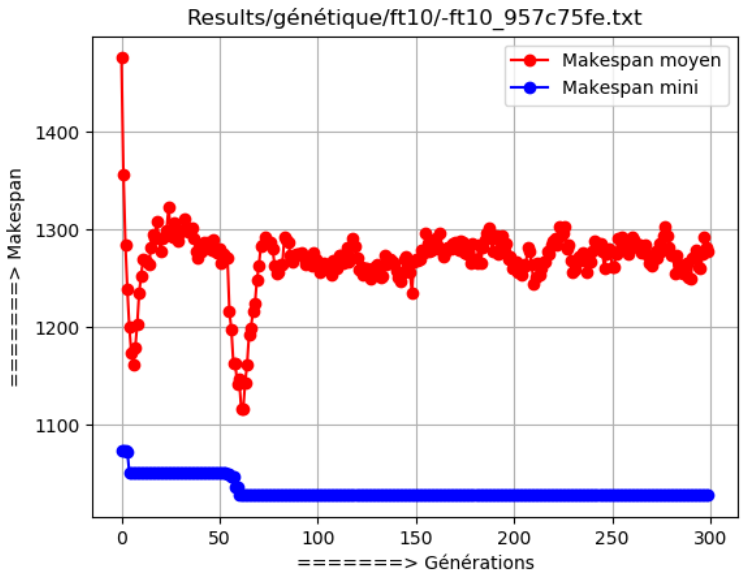


FIGURE 5 – Evolution du makespan moyen et mini d'une population au cours des générations (sur ft10).

nom	nombre d'instance	makespan moyen	temps moyen de convergence (s)	makespan mini	Optimum
ft06	20	56.75	13.768	55	55
ft10	20	1058.85	35.11	1017	930
la01	24	678	86.79	666	666

TABLE 5 – Résultats algorithme génétique.

6 Conclusion

Nous avons vu différentes méthodes pour résoudre un problème de planification de type “jobshop”. La méthode exacte via la librarie docplex d’IBM offre les résultats les plus satisfaisants en terme de temps et de solution trouvée. En effet les algorithmes de recherche locale et génétique, développés, sont moins performants. Le tableau 6 résume les performances de chaque algorithme.

		Méthode exacte	Heuristique gloutonne	Recherche Voisinage	Génétique
ft06	makespan	55	79	55	55
	temps	<0.01	<0.01	88	14
ft10	makespan	930	1075	997	1017
	temps	1.71	<0.01	792	35
ft20	makespan	1165	1268	1267	1267
	temps	0.32	<0.01	392	>600
la01	makespan	666	771	697	666
	temps	<0.01	<0.01	89	87

TABLE 6 – Résultats algorithme génétique.

A Heuristique gloutonne instance “ta“

nom	SPT	LPT	Optimum
ta01	1464	1738	1231
ta02	1451	1758	1244
ta03	1495	1656	1218
ta04	1709	1800	1175
ta05	1618	1893	1224
ta06	1520	1683	1238
ta07	1435	1773	1227
ta08	1458	1577	1217
ta09	1855	1747	1274
ta10	1700	1779	1241
ta13	1843	2163	None
ta14	1638	1787	1345
ta15	1946	2114	None
ta17	2005	2080	1462
ta18	1782	1992	None
ta24	2076	2702	None
ta31	2553	2430	1764
ta32	2405	2537	None
ta35	2396	2781	2007
ta36	2367	2660	1819
ta37	2665	2518	None
ta38	2192	2479	1673
ta39	2442	2640	1795
ta43	2393	2792	None
ta49	2706	3156	None
ta50	2486	2703	None

TABLE 7 – Makespan de l’algorithme glouton en fonction de différentes priorités (instances de type ’ta’).

B Heuristique gloutonne instance “la“

nom	SPT	LPT	Optimum
la01	771	822	666
la02	823	991	655
la03	707	825	597
la04	713	820	590
la05	614	621	593
la06	1203	1125	926
la07	1036	1070	890
la08	944	1037	863
la09	1045	1186	951
la10	1051	1132	958
la32	2165	2440	1850
la33	1904	2297	1719
la34	2008	2333	1721
la35	2164	2350	1888
la36	1840	1761	1268
la37	1656	1944	1397
la38	1611	1732	1196
la39	1501	1625	1233
la40	1476	1822	1222

TABLE 8 – Makespan de l’algorithme glouton en fonction de différentes priorités (instances de type ’la’).

C Pseudo codes, heuristique gloutonne.

Algorithm 1 Pseudo code : Heuristique gloutonne

Require: $Instance$: $object$
Require: $Jobs$: $object$
Require: $task$: $object$
  $t \leftarrow 1$
  while $t < 0$ **do**
    if $Instance$ is $finish$ **then**
      $BREAK$
    end if
    for $Ressource$ in $Instance$ **do**
      if $Ressource$ not $allocated$ **then**
        CALL $recherche\_tache(t)$
      end if
      if Current task is $finish$ **then**
        $Desallocate\ Ressource$
        $Update\ Jobs\ states$
        $Update\ Instance\ states$
        CALL $recherche\_tache(t)$
      end if
    end for
    $t \leftarrow t + 1$
  end while
  return Makespan

Algorithm 2 Pseudo code : Recherche tache

Require: *Instance* : *object*
Require: *Jobs* : *object*
Require: *task* : *object*
  possible_task ← []
  for each jobs of the instance **do**
    if *Job* is not *finish* **then**
      if *Current job Ressource* is *available* **then**
        possible_task ← *Current job Ressource*
      end if
    end if
  end for
  for each *possible_task* **do**
    find the one corresponding to the heuristic (LPT or SPT, ie min or max search)
    task ← *correspondance*
  end for
  Allocate task to resource
  Update Jobs states
  Update Instance states

D Pseudo codes, recherche locale.

Algorithm 3 Pseudo code : exploration voisinage

Require: $Solution : object$
Require: $n : int$, nombre de voisin
Require: $max_{depth} : int$, profondeur de recherche
Require: $crit\_stagnation : int$, compteur de stagnation

```
to_be_explored ← Solution_voisins
explored ← []
best ← Solution_sequence
opti ← Solution_makeSpan
depth ← 0
alert ← 0
compt ← 0
while to_be_explored AND depth < max_depth do
  if alert = crit_stagnation then
    BREAK
  else
    depth ← to_be_explored[0]_depth
    to_be_explored, explored, make_min, seq ← explore_deeper()
  end if
  if make_min < opti then
    alert ← 0
    opti ← make_min
    best ← seq
  else
    alert ← alert + 1
  end if
  compt ← alert + 1
end while
```

Algorithm 4 Pseudo code : explore deeper

Require: $depth : int$
Require: $to\_be\_explored : ListofSolution$
Require: $explored : ListofSolution$
Require: $n : int$
  $origine \leftarrow to\_be\_explored[0]$
  $to\_be\_explored \rightarrow pop[origine]$
  $explored \leftarrow origine$
  $best\_voisin \leftarrow origine_{best\_voisin}[: 2]$
  $make \leftarrow []$
  for $voisin$ in $best\_voisin$ **do**
    if $voisin$ not in $explored$ **then**
      $to\_be\_explored \leftarrow voisin$
      $ecart \leftarrow (origne_{makeSpan} - voisin_{makeSpan})/origine_{makeSpan}$
    else
      $ecart \leftarrow 0$
    end if
    $make \leftarrow voisin_{makeSpan}$
  end for
  RETURN $to\_be\_explored$, $explored$, $min(make)$, $voisin$

E Pseudo codes, génétique.

Algorithm 5 Pseudo code : main génétique

Require: $nombre\_de\_generation : int$
Require: $population : Object$
  CALL $genese()$
  while $population_{generation} <= nombre\_de\_generation$ AND $temps < 900$ AND $alert < nombre\_de\_generation * 0.3$ **do**
    CALL $evaluation()$
    CALL $selection()$
    CALL $croisement()$
    CALL $evaluation()$
    if $nouveloptimal$ **then**
      $alert \leftarrow 0$
    else
      $alert \leftarrow alert + 1$
    end if
  end while

Algorithm 6 Pseudo code : genese

Require: $ratio\_voisin : float$
Require: $n : int$
Require: $population : Object$
  $sequence \leftarrow []$
  $sequence \leftarrow +heuristique\_glounne\_SPT$
  $sequence \leftarrow +heuristique\_glounne\_LPT$
  for i in $range(n * ratio\_voisin)$ **do**
    $rnd \leftarrow randint(0, 2)$
    if $rnd > 1$ **then**
      $sequence \leftarrow +permutation\_SPT$
    else
      $sequence \leftarrow +permutation\_LPT$
    end if
  end for
  $sequence \leftarrow +permutation\_random * (n - len(sequence))$
  for s in $sequence$ **do**
    ADD s à la population $population$
  end for

Algorithm 7 Pseudo code : evalution

Require: $population : Object$
  for $individu$ in $population$ **do**
    Calcul du makespan
  end for
  Calcul des probabilités
  Calcul du facteur de mutation

Algorithm 8 Pseudo code : selection

Require: $population : Object$
Require: $n : int$
  if Sélection par roulette **then**
    Tirage aléatoire de n individu selon leur probabilité
    Ajout de 0.1 * n "élite de la population"
  end if
  if Sélection par élitisme **then**
    Trie des individus selon leur makeSpan
    Tirage des n premier individu trié
    Remplacement de la population
    Calcul des probabilités
  end if

Algorithm 9 Pseudo code : croisement

Require: $population : Object$
Require: $beta : Double 0 <=:<= 1, facteurdemutation$
Require: $alpha : Double 0 <=:<= 1, probadecroisement$
Require: $n : int$
  if Croisement par roulette **then**
    while $nombre\_enfant <= n * alpha$ **do**
      Tirage aléatoire de 2 individus selon leur probabilité.
      Croisement selon un opérateur de croisement
      Pour chaque enfant(2) : CALL mutation
    end while
  end if
  if Croisement par duel **then**
    while $nombre\_enfant <= n * alpha$ **do**
      Tirage aléatoire de 2 pères selon leur probabilité.
      Sélection du meilleur
      Tirage aléatoire de 2 mères selon leur probabilité.
      Sélection de la meilleure
      Croisement selon un opérateur de croisement
      Pour chaque enfant(2) : CALL mutation
    end while
  end if

Algorithm 10 Pseudo code : mutation

Require: $population : Object$
Require: $enfant : Objectindividu$
Require: $beta : Double 0 <=:<= 1, facteurdemutation$
  $beta \leftarrow beta * (1 + population_{facteurMutation})$
  $rnd \leftarrow random()$
  if $rnd < beta$ **then**
    $enfant_{sequence} \leftarrow mutation\_par\_inversion$
  end if
  $rnd \leftarrow random()$
  if $rnd < beta$ **then**
    $enfant_{sequence} \leftarrow mutation\_par\_insertion$
  end if
  $rnd \leftarrow random()$
  if $rnd < beta$ **then**
    $enfant_{sequence} \leftarrow mutation\_par\_permutation$
  end if
  $rnd \leftarrow random()$
  if $rnd < beta$ **then**
    $enfant_{sequence} \leftarrow mutation\_par\_decalage$
  end if

Références

[1] Anthony Caumond. "Le problème de jobshop avec contraintes : modélisation et optimisation". Thèse de doct. Université Blaise Pascal - Clermont-Ferrand II, 2006.

[2] José Gonçalves, Jorge Mendes et Mauricio Resende. "A Hybrid Genetic Algorithm for the Job Shop Scheduling Problem". In : *European Journal of Operational Research* 167 (fév. 2005), p. 77-95. doi : 10.1016/j.ejor.2004.03.012.

[3] Ye Li et Yan Chen. "A Genetic Algorithm for Job-Shop Scheduling." In : *JSW* 5 (jan. 2010), p. 269-274.

[4] Mahanim Omar, Adam Baharum et Yahya Hasan. "A JOB-SHOP SCHEDULING PROBLEM (JSSP) USING GENETIC ALGORITHM (GA)". In : (jan. 2006).

[5] Ren Qing-Dao-Er-Ji et Yuping Wang. "A new hybrid genetic algorithm for job shop scheduling problem". In : *Computers Operations Research* 39 (oct. 2012). doi : 10.1016/j.cor.2011.12.005.