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

Auteur :
M. Pierre-Arnaud Auque

Version 1.0 du
13 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 sera 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.

— Dans un premier temps un algorithme glouton.
— Dans un second temps un algorithme de recherche locale.
— Pour finir par un algorithme génétique.

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. Il est donc plus facile, à partir de cette représentation d'effectuer des permutations et donc des recherches de voisinages.

2 Algorithme Glouton

2.1 Heuristique générale

L'algorithme glouton consiste à allouer des tâches aux différents machines, en fonction d'une règle de priorité. Les règles implémentés 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).

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

TABLE 1 – Makespan de l'algorithme glouton en fonction de différentes priorités (instances de type 'ft').

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

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 3 – Makespan de l'algorithme glouton en fonction de différentes priorités (instances de type 'ta').

2.2 Heuristique randomisée

L'Heuristique randomisée se base sur l'Heuristique gloutonne, en ajoutant une part d'aléatoire dans le choix des tâches à allouée. Dans la table **??**, 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 gloutonne, 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 4 – Algorithme glouton sur instances de type 'ft', rnd = 1 correspond à une heuristique entièrement randomisée.

3 Recherche locale

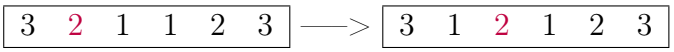
La recherche locale, consiste à explorer les voisinages d'une solution donnée. Les métriques d'évalutions sont le makespan et la durée de recherche du voisinage (ou nombre d'itération). Les paramètres de cette recherche locale sont :

— 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.

3.1 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.

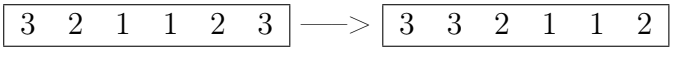


Le code ci-dessous permet à partir d'une liste, de générer l'ensemble des permutations deux à deux possible :

```
def permutation_by_2_all(liste):
    liste_permut = []
    for j in range(0, len(liste) - 1):
        for i in range(j, len(liste) - 1):
            tempo = copy.copy(liste)
            temp = tempo[j]
            tempo[j] = tempo[i + 1]
            tempo[i + 1] = temp
            liste_permut.append(tempo)
    return liste_permut
```

3.2 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.

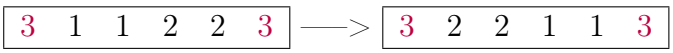


Le code ci-dessous permet à partir d'une liste, de générer l'ensemble des décalages possibles :

```
def decalage_all(liste):
    liste_decale = []
    for j in range(0, len(liste)):
        tempo = copy.copy(liste)
        liste_decale.append(np.roll(tempo, j))
    return liste_decale
```

3.3 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.




```
def inversion_all(liste):
    liste_inverse = []
    for i in range(0, len(liste)):
        for j in range(len(liste), i + 1, -1):
            tempo = copy.copy(liste)
            tempo_head = tempo[:i]
            to_be_flipped = tempo[i:j]
            tempo_tail = tempo[j:]
            liste_inverse.append(np.concatenate((tempo_head,
                                                 np.flip(to_be_flipped)
                                                 tempo_tail)))
    return liste_inverse
```

3.4 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 **??**. 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

TABLE 5 – Résultats de la recherche de voisinage sur les instances de types ft.

4 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 génétiques on été dévéloppée. 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.

4.1 Gènése

La génése est la création de la population initiale. Cette population est crée aléatoirement via un vecteur à répétition, l'heuristique gloutonne (SPT et LPT) fait partie de cette population initiale. La taille de la population initiale est un hyperparamètre important. Ici des classes supplémentaires sont créés, l'individu et la population.

```
def genese(n, population):
    # Generation de la population initiale
    # n , un entier, qui defini la taille de la population
    # population, un objet Population
    inst = Instance(population.nom)

    # On ajoute le SPT et LPT    la population initiale et
       des permutations de SPT
    heuristique_gloutone(inst, verbose=0, prio="SPT", rnd=0)
    liste = vecteur_bier(inst)
    sequence = permutation_random(liste, n - 2)
    sequence.append(liste)
    heuristique_gloutone(inst, verbose=0, prio="LPT", rnd=0)
    liste = vecteur_bier(inst)
    sequence.append(liste)

    # Instance des individus selon la s quence
    for i, seq in enumerate(sequence):
        indv = Individu(population, "Adam", "Eve", seq)
        population.add_Indv(indv)
```

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

```
def evaluation(population):
    # Evaluation d'une population (fitness et makeSpan), et
       mis    jour des probabilite de selection
    #   et du facteur de mutation
    # population, un objet Population

    for indv in population.individu:
        indv.set_Cout(alloc_avec_liste(Instance(population.nom),
           indv.sequence))
    population.calc_Proba_rg()
    population.set_facteurMutation()
```

4.3 Croisement

Le croisement d'individus va permettre d'obtenir la génération suivante. Deux individus sont tirés aléatoirements (selon les probabilités définies lors de l'évaluation) dans la population, afin de déterminer le père et la mère lors du croisement. Ainsi 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. Chaque croisement genère deux enfants différents (l'un qui va plus hérité de la mère et l'autre plus du père).

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

4.4 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 à 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. Ainsi les chances de mutations doublent toute les 10 générations sans changement.

On distingue quatres opérateurs de croisement :

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

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

Lees quatres opérateurs de mutations peuvent se produire sur un même enfant.

```
def mutation(enfant, population, beta=0.15):
    beta = beta * (1 + population.facteurMutation)
    # Mutation de l'enfant sur chaque gene, mutation
       al atoire

    # mutation par inversion
    rnd = random.random()
    if rnd < beta:
        rnd1 = np.random.choice(range(len(enfant.sequence)),
           size=2, replace=False)

        tempo = copy.copy(enfant.sequence)
        tempo_head = tempo[:min(rnd1)]
        to_be_flipped = tempo[min(rnd1):max(rnd1)]
        tempo_tail = tempo[max(rnd1):]
        enfant.sequence = np.concatenate((tempo_head,
           np.flip(to_be_flipped), tempo_tail))

    # mutation par insertion
    rnd = random.random()
```

```
if rnd < beta:
    rnd1 = np.random.choice(range(len(enfant.sequence)),
       size=2, replace=False)

    tempo = copy.copy(enfant.sequence)
    tempo = np.delete(tempo, max(rnd1))
    enfant.sequence = np.insert(tempo, min(rnd1),
       enfant.sequence[max(rnd1)])

# mutation par permutation
rnd = random.random()
if rnd < beta:
    rnd1 = np.random.choice(range(len(enfant.sequence)),
       size=2, replace=False)
    temp = enfant.sequence[min(rnd1)]
    enfant.sequence[min(rnd1)] =
       enfant.sequence[max(rnd1)]
    enfant.sequence[max(rnd1)] = temp
    enfant.set_Mutation((min(rnd1), max(rnd1)))

# mutation par d calage
rnd = random.random()
if rnd < beta:
    rnd1 = np.random.choice(range(len(enfant.sequence)),
       size=1, replace=False)
    tempo = copy.copy(enfant.sequence)
    tempo = np.roll(tempo, rnd1)
    enfant.sequence = tempo
    enfant.set_Mutation((min(rnd1), max(rnd1)))
```

4.5 Sélection

L'étape de sélection va 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.

```
def selection(population, n):
    # Selection des meilleurs enfants, selection par roulette
    temp_proba = []
    for indv in population.individu:
        temp_proba.append(indv.proba)

    tirage = np.random.choice(population.individu, size=n,
       replace=False, p=temp_proba)
    population.reset_Indv()

    # Ajout des individus choisi
    for indv in tirage:
        population.add_Indv(indv)
    # On ajoute 10% d'elite a la selection
    for i in range(int(n * 0.10)):
        indv = copy.copy(population.elite)
        indv.generation = population.generation
        population.add_Indv(indv)

    population.calc_Proba_rg()
```

4.6 Résultats

L'algorithme génétique a été testé sur les instances ft , la01, et ta01.

nom	nombre d'instance	makespan moyen	temps moyen de convergence (s)	makespan mini	Optimum
ft06	20	55.10	31.1775	55	55
ft10	19	1055.31	15.07	1014	930
ft20	4	1267	250	1267	1165
la01	35	678	52	666	666
ta01	8	1449	4	1402	231

TABLE 6 – Résultats algorithme génétique.

5 Annexes