Projet PIMA

Allocation de tâches à des machines avec critère égalitarien

Etudiants :
David Sarmiento
Félicité Lordon
Shuting Zhang

Encadrante :
Fanny Pascual

9 septembre 2018

Table des matières

Introduction

Un des problèmes fondamentaux de l'ordonnancement est l'allocation de n tâches à $m > 2$ machines parallèles pour minimiser le temps de complétion ou makespan.

La quantité à minimiser que l'on appellera makespan ou coût-max peut être défini de plusieurs manières, selon le problème. Les deux principaux critères utilisés en pratique sont :

1. le critère égalitarien : on cherche à minimiser le coût de la tâche la plus importante.
2. le critère utilitarien : on cherche à minimiser la somme totale des coûts.

Il s'agit d'un problème qui intéresse les usines et centres de données, qui veulent satisfaire au mieux le client le moins bien servi (choix égalitarien) ou économiser leurs ressources au détriment des clients les moins bien desservis (choix utilitarien) lors de l'allocation de tâches de tailles diverses à des machines. Le second critère étant déjà très étudié, nous nous intéresserons au premier.

Cependant, les tâches y sont souvent hétérogènes et en compétition pour différentes ressources. Par exemple, dans un centre de données, une machine peut héberger des tâches gourmandes en mémoire, bande-passante ou calculs. Il est donc crucial de prendre en compte, dans la définition du coût d'une tâche, non seulement de sa taille, mais aussi des ressources qu'elle convoite avec d'autres tâches, c'est-à-dire de sa compatibilité avec les autres tâches. Pour cela, on introduit la notion de types. Chaque tâche est d'un certain type, et chaque type possède un certain coefficient de compatibilité avec chaque autre type.

Dans un premier temps, on examinera les algorithmes classiques d'approximation qui ont été proposés dans la littérature pour résoudre ce problème de l'allocation de tâches caractérisées par leur taille et type, avec critère égalitarien.

Dans un second temps, on exposera deux schémas d'approximation. En effet, il s'agit aussi d'un problème NP-difficile au sens fort, c'est-à-dire qu'il reste NP-difficile même lorsque la taille des données d'entrée (nombre de machines, de tâches et matrice de types) est bornée par un polynôme. A moins que P=NP, ces schémas d'approximation ne peuvent pas être complètement polynomiaux.

Enfin, on présentera quelques heuristiques que nous avons inventées.

Ces différents algorithmes feront l'objet de multiples comparaisons pour analyser leur performance dans différentes classes d'instance.

Méthodes et outils

Notations

Dans la suite, on notera :

1. m le nombre de machines, n le nombre de tâches
2. $types$ la liste de types disponibles
3. $MCoeff = M\alpha_{i,j}, (i,j) \in types$ la matrice des coefficients de compatibilité entre les types
4. $\alpha_{i,j}$ le coefficient de comptibilité entre les types i et j
5. LM une liste de m machines, LT une liste de n tâches aux types renseignés par $MCoeff$
6. p_i la taille de la tâche i, t_i le type de la tâche i

Par ailleurs, voici le sens que l'on donnera au makespan et au coûtMax d'une liste de machines LM avec $m > 0$:

$$\boxed{makespan = \max_{mach \in LM} \sum_{i \in mach} p_i} \quad \boxed{coutMax = \max_{mach \in LM} \sum_{i,j \in mach} p_i \times \alpha_{i,j}}$$

Remarques :

1. $makespan = coutMax$ si les tâches sur la machine sont de même type, donc a fortiori, si les tâches sont d'un seul type
2. $\alpha_{i,j} > 1$ pour des types i, j incompatibles, $\alpha_{i,j} < 1$ pour des types i, j compatibles et $\alpha_{i,i} = 1$
3. La longueur d'une machine est définie comme la somme des longueurs des tâches sur cette machine.

Implémentation et Evaluation des algorithmes

Pour appliquer le critère égalitarien à la comparaison des algorithmes, nous utilisons le makespan (mks) dans le cas ou les tâches ne sont que d'un type et le coût-max lorsqu'il y a plusieurs types.

La moyenne de ces mesures pour un nombre suffisant d'exécution des algorithmes permet de les comparer. Ces différenctes exécutions ne sont pas totalement aléatoires, car la plupart des algorithmes donnent des résultats significativement différents selon la "classe d'instance" sur laquelle ils sont testés. Une classe d'instances correspond à un ensemble de contraintes sur les paramètres de test. Par exemple, les coefficients de $MCoeff$ peuvent tous être dans un certain intervalle, le ratio n/m et le nombre de types peuvent varier.

Les paramètres auxquels nous nous sommes intéressés concernent le nombre de tâches pour un nombre de machines fixé, le nombre de types, et la variabilité des coefficients de compatibilité entre les types.

Nous avons utilisé la bibliothèque numpy et des dictionnaires comme structures de données.

Une interface graphique réalisée avec $Tkinter$ nous a permis de visualiser l'allocation des tâches pour chaque algorithme. Elle a aussi générer les images d'allocation qui paraissent dans ce rapport. La bibliothèque $Matplotlib$ nous a permis de comparer les algorithmes.

1 Algorithmes classiques d'allocation

1.1 Largest Processing-Time first (LPT)

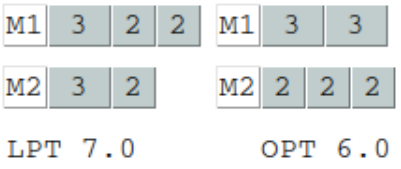
LPT consiste à allouer les tâches, dans l'ordre inverse de leur taille, aux machines les moins chargées :

Algorithm 1: LPT

Entree: LT triée à allouer à LM vide
Sortie : LM allouée
Corps :
  for *i=1 à len(LT)* **do**
    $LT[i]$ est allouée à la machine de LM la moins chargée
  return LM

LPT est en $O(n*m)$ ce qui est très efficace par rapport aux algorithmes qui suivent. Cependant, il ne prend pas en compte le type, et donne souvent des résultats éloignés de la solution optimale. Même avec des tâches d'un seul type, LPT n'est pas forcément optimal (voir figure 1). Il es évident que LPT est optimal est lorsque $n < m$.

Figure 1 – cout de LPT et OPT pour des tâches d'un seul type



Supposons que $n > m$ et cherchons à évaluer l'approximation de l'allocation LPT par rapport à l'allocation OPTimale.

Lemma 1.1. *On définit LB (lower-bound) comme le maximum de la plus grande tâche et de la moyenne de la longueur des tâches :* $LB = \max(p_{max}, \frac{\sum_{i=1,n} p_i}{m})$ *On a la relation* $LB <= MakespanOpt <= 2 * LB$

Démonstration. D'après la définition du makespan, il ne peut être plus petit que p_{max} la taille de la plus grande tâche, ou de la moyenne des longueurs des tâches. Donc $LB <= MakespanOpt$

D'une part, si $LB = p_{max}$. On place la tâche de taille LB sur une machine M. Comme le poids de M est plus grand ou égal à la moyenne et à toutes les tâches restantes, on peut remplir LM pour que $MakespanOpt <= 2 * LB$ D'autre part, si $LB = \frac{\sum_{i=1,n} p_i}{m}$. On place la tâche de taille p_{max} et $< \frac{\sum_{i=1,n} l_i}{m}$ sur une machine M. D'après ces 2 inégalités, on peut répartir les autres tâches de tailles inférieures sur LM pour que $MakespanOpt <= 2 * LB$

□

Theorem 1.2. *LPT est 1/2-approché*

Démonstration. Soit le makespan le coût de la machine i la plus chargée. Soit $p_{min} = p_n$ la dernière tâche ajoutée à i. Avant son ajout, i était la machine la moins chargée. On a donc $L_i - p_{min} < L_k \forall k \in 1, ..., m$ d'ou $L_i - p_{min} < \frac{\sum_{i=1,n} p_i}{m}$ d'ou $L_i - p_{min} < OPT$ d'après le lemme 1 On a donc $Mksp = L_i - p_{min} + p_{min} < OPT + p_{min} * OPT$ On cherche à présent à majorer p_{min} en fonction de OPT. On a $2 * p_{m+1} < 2 * p_n < OPT$ puisque la machine qui contient p_{m+1} en contient forcément une autre plus grande. Donc $p_{min} < 1/2 * OPT$ et $Mksp < OPT + 1/2 * OPT$

En conclusion, on a $Mksp = L_i - t_i + t_i$ ¡ OPT + $1/2 * OPT$ et LPT est 1/2-approché

Remarque : Si l'on a qu'un seul type, LPT est même 1/3-approché : On a soit $p_{min} < 1/3 * OPT$, soit $p_{min} > 1/3 * OPT$ Dans le premier cas, on a $Mksp < OPT + 1/3 * OPT$ Dans le second cas, LPT est optimal. En effet, comme $OPT < 2 * p_{min}$, au plus 2 machines sont allouées sur chaque machine, et l'allocation LPT est optimale. □

Les algorithmes qui vont dorénavant être exposés prennent en compte la gestion des types, souvent avec LPT comme point de départ, au prix d'une moins bonne complexité,

1.2 Scheduled-juxtaposed

Lorsqu'il n'y a que deux types, Juxtaposed peut être utilisé. Il consiste à séparer les tâches de types 1 et de type 2 en deux groupes. On essaie d'allouer les tâches 1 à un nombre de machine i compris entre 1 et m, et les tâches 2 à m - i machines. On retient le coutMax de cette allocation. Puis l'on cherche le nombre i de machines tel que le couMax soit minimisé.

Une fois que le nombre de machines pour chaque type est déterminé, Juxtaposed fait appel à un algorithme ALLOC pour réaliser l'allocation, qui peut être LPT par exemple. La complexité de juxtaposed dépend de $C\_ALLOC(k, n)$, la complexité de ALLOC pour k machines et n tâches est en $O(n + \sum_{k=1}^{m} C\_ALLOC(k, n))$.

Algorithm 2: Alloc-Scheduled-Juxtaposed

Entree: LT à allouer à LM vide, ALLOC un algorithme d'allocation, par exemple LPT ou PTAS (expliqués plus loin)

Sortie : LM allouée

// initialisation
$LT1$ taches de type 1
$LT2$ taches de type 2
Cost1, Cost2 listes vides

// remplissage des listes $Cost1$ et $Cost2$
for $i = 0$ à $i = m$ - 1 **do**
    // allocation des taches de type 1 sur les i premières machines
    $LM1 \leftarrow ALLOC(LM[:i], LT1)$
    $CoutMax(LM1)$ est ajouté à Cost1
    // allocation des taches de type 2 sur les m - i dernières machines
    $LM2 \leftarrow ALLOC(LM[i:], LT2)$
    $CoutMax(LM2)$ est ajouté à Cost2

// allouer 2 groupes de machines selon le meilleur indice de partition
$indPartition \leftarrow$ ind tel que $max(Cost1[ind], Cost2[ind])$ soit minimisé)
$LMind \leftarrow$liste de machines de taille $indPartition$
$LMcompl \leftarrow$liste de machines de taille $(n - indPartition)$
$LM1Finale \leftarrow$ ALLOC(LMind, LT1)
$LM2Finale \leftarrow$ ALLOC(LMcompl, LT2)
$LM \leftarrow LM1Finale + LM2Finale$

return LM

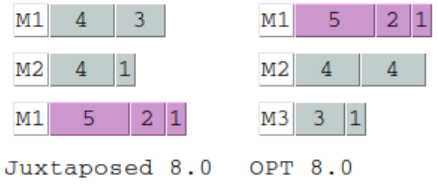


Figure 2 – types incompatibles (2)

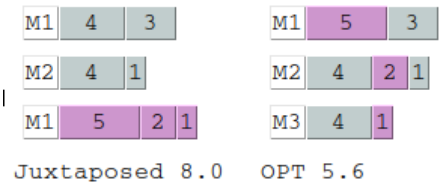


Figure 3 – types compatibles (0.2)

Il est très efficace lorsque les deux types ne sont pas compatibles puisqu'il assure leur séparation, mais moins lorsque les types sont compatibles puisqu'il tend à disposer sur une même machine des tâches de même type dont le coefficient partagé est $1 > 0$.

2 Deux Schémas d'approximation

Un schéma d'approximation est un algorithme qui pour tout $\epsilon > 0$ et toute instance I du problème d'allocation, produit, en un temps polynomial en $|I|$, une solution telle que : $OPT(I) < PTAS(I) < (1 + \epsilon) * OPT(I)$

2.1 Une application du problème du Bin-Packing

2.2 Le PTAS

Le schéma d'approximation consiste à allouer les k plus grandes tâches de façon optimale, puis à appliquer LPT sur les $n - k$ tâches restantes. Leur poids faible limite en effet l'influence de leurs coefficients qui ne sont pas pris en compte.

Pour allouer les k plus grandes tâches de manières optimales on effectura une récursion sur k, et à chaque appel une boucle sur m sera faite. On peut le voir comme un arbre non binaire, composé de k niveaux, où chaque noeuds possèdent m fils. En effet, pour allouer les k plus grandes tâches de manières optimales il faut tester toutes les possibilités d'allocation. Pour chaque k plus randes tâches, on aura donc m choix. Ainsi, un chemin jusqu'à une feuille de l'arbre représente une instance d'allocation des k tâches. On notera un tel chemin ChemF. L'arbre ne sera pas généré entièrement, on ne gardera en mémoire que le ChemF ayant un coût max minimal. Enfin, pour accélérer l'algorithme on appliquera une méthode de séparation et évaluation (branch and bound method). On arrêtera alors de construire la branche s'il s'avère que son coût max dépasse déjà celui de ChemF.

La complexite de cet algorithme est en O(m**k), d'où le fait que l'on restreint à k autant que possible.

Algorithm 3: rec_PTAS_k_tasks

Entree: LM vide, LT triée par taille décroissante,
Sortie : LM allouée

```
// Branch-and-Bound method
cost = CoutMax(LM)
if cost >= curentLowestCostMax then
        // coût de cette branche supérieur au plus bas, on coupe
        return

if k == 0 then
    // toutes les k tâches ont été allouées
    if cost! = 0 then
        currentLowestCostMax ← cost
        currentBestLM ← LM
    return

// allouer la première tâche de LT à la meilleure machine entre toutes
for indMachine from 0 to m-1 do
    if len(LT) then
        tmpTask ← LT[0]
        ajouter LT[0] à la machine courante
        retirer LT[0] de LT
        // modification de LM si la machine courante permet de minimiser currentLowestCostMax
        rec_PTAS_k_tasks(LM, LT, k-1, currentLowestCostMax, currentBestLM)
        // remise à jour de LT et de la machine courante après modification potentielle de LM
        Insérer tmpTask au début de LT
        Retirer tmpTask de la machine courante
```

Comme pour Scheduled-Juxtaposed, on utilise un algorithme ALLOC (par exemple, LPT ou LPT_typed) pour allouer les n-k tâches restantes : La complexité de PTAS est en O(m**k + Complexite(ALLOC)) Il faut donc bien faire attention au choix de k, puisque la complexité exponentielle peut vite l'emporté par rapport à Complexite(ALLOC)

3 Recherche de nouvelles heuristiques

Nous allons à présent nous intéresser à des algorihtmes qui prennent en compte plus de deux types.

3.1 un algorithme inspiré de LPT

L'algorithme agit comme lpt mais prend en compte les types. Pour ce faire, la tâche a allouer ne sera mise pas sur la machine la moins chargée mais sur celle où elle l'affectera le moins. On alloue temporairement la tâche à toutes les machines une par une, en calculant son coût max. La tâche sera ensuite alloué sur la machine où après ajout, le coût max est minimum. i.e. ce n'est pas la différence entre coût max avant et après ajout qui est pris en compte.

Algorithm 4: LPT-Typed

Entree: LM vide, LT triée par taille décroissante, group = False
Sortie : LM allouée

// initialisation
for *machine in LM* **do**
    Ajouter *tache* la plus grosses des tâches de LT dans machine
    Retirer *tache* de LT

for *tache in LT* **do**
    Soit C une liste vide
    for *machine in LM* **do**
        Ajouter *tache* à *machine*
        Stocker le coût max pour cette configuration dans la liste C
        Retirer *tache* de *machine*
        On regarde pour quelle machineBest de LM le coût est minimum parmi les coûts dans C
        On ajoute *tache* à *machineBest*, et on la retire de LT
        Si *group*, alors on groupe toutes les tâches en fonction de leurs types dans la liste de machine
        // cette option permet d'accélérer les calculs du CoutMax

return LM

La complexité de cet algorithme est de O(n*(m*n + m)) assez efficace par rapport aux résultats obtenus. Des variantes de lpt-typed ont été faites afin de l'améliorer, la version présente est une combinaison de plusieurs autres. Néanmoins on peut noter la version lpt-typed-difference, où justement l'on prend la différence minimale entre le coût max avec et après ajout.

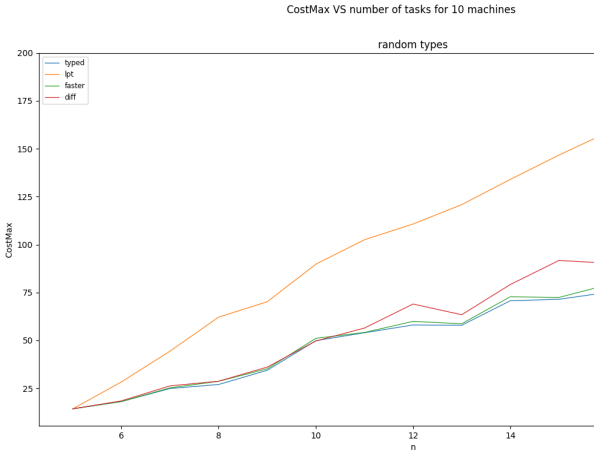


Figure 4 – CoutMax en fonction du nombre de tâches pour différentes versions de LPT

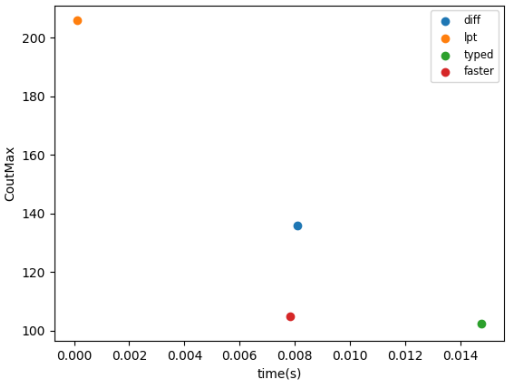


Figure 5 – CoutMax en fonction du tmps d'execution pour différentes versions de LPT

3.2 des algorithmes sensibles au regroupement des types

D'autre part, certaines heuristiques consistent à rassembler au maximum sur une même machine des tâches plus compatibles entre elles qu'avec d'autres. Une fois que l'on a déterminer quelle tâche peut cohabiter avec quelle tâche, il y a un équilibre à atteindre entre le nombre de ces tâches allouées à une même machine et leur réelle compatibilité, qui doit permettre de minimiser le coût-max général :

Ce coût-max optimal relativement au regroupement des tâches, et est déterminé par dichotomie.

Pour cela, on détermine un nouvel encadrement pour le coutMax OPT, en étendant l'encadrement trouvé pour le Makespan (lemme 1.1)

Lemma 3.1. *Soit* $LB_{type} = \max(p_{max}, \max_{t \in types} \frac{\sum_{taskdetypet} p_{task}}{m})$

On a alors $LB_{type} < CoutMaxOpt < 2 * LB * max_{MCoeff}$

Démonstration. Inégalité 1 : Le Cout d'une machine est supérieur à la somme des longueurs des tâches d'un type sur cette machine. Le CoutMax est donc supérieur au maximum de la somme des longueurs des tâches d'un même type pouvant être sur une même machine. Ce maximum est LB_{type}, ce qui prouve la première inégalité. Attention, même si on a $\frac{\sum_{i=1,n} p_i}{m} < LongueurMax$ on n'a pas forcément $\frac{\sum_{i=1,n} p_i}{m} < CoutMaxOpt$ puisque les coefficients de compatibilité entre les tâches peuvent être < 1. C'est pourquoi il était nécessaire d'introduire LB_{type}

Inégalité 2 : De même, même si la longueur maximale d'une machine est toujours majorée par 2 * LB comme lorsqu'il n'y a qu'un type, ce n'est pas le cas du coûtMaxOpt. En effet, celui-ci est calculé en fonction du coefficient de compatibilité entre les tâches cohabitant sur une machine qui peut être > 1 . Cependant, on peut remarquer que le CoutMax d'une machine chargée d'une longueur L est majoré par $L * max_{MCoeff}$. Donc le CoûtMax d'une machine de longueur maximale 2 * LB est majoré par $2 * LB * max_{MCoeff}$, ce qui prouve la seconde inégalité. □

Une fois que l'on a un encadrement pour le CoutMax, on fixe arbitrairement une *precision* à atteindre pour le trouver par dichotomie ($precision = 0.001$ par exemple). Puis l'on teste si notre algorithme peut allouer avec succès toutes les tâches avec un CoutMax fixé à la moitié de l'intervalle, en passant à une nouvelle machine lorsque le CoutMax visé est dépassé. Selon le résultat de l'expérience, on peut restreindre de moitié l'intervalle et répéter l'opération jusqu'à obtenir la précision souhaitée. On aura alors trouvé le meilleur CoutMax possible pour cet algorithme et cette précision. L'execution de l'algorithme avec ce meilleur CoutMax possible donne la véritable allocation finale.

Remarque : Si on note Δ l'intervalle de dichotomie, le nombre d'appels K à l'algorithme est tel que $\Delta < 2^K * precision$. Il y donc a au plus $\lceil log_2(\frac{\Delta}{precision}) \rceil$ appels à l'algorithme, ce qui est indépendant des variables d'entrée.

3.3 Greedy_Cluster : Clusterisation des types

La première heuristique au regroupement des types est Greedy-Cluster.

L'algorithme *Rec_Clusterize* se chargera réunir les tâches par cluster où tous seront des types compatibles entre eux au sein de ces clusters.

Algorithm 5: Rec_Clusterize / Complexité : O(Ntypes !)

Entree: ListCluster liste de clusters à remplir, AllTypes
Sortie : ListCluster remplie

if *len(AllTypes) == 0* **then**
    return ListCluster

Si vide, initialisation du cluster ListCluster[0] avec le premier type de AllTypes qui en est retiré

// remplissage du cluster
for *current_type in AllTypes* **do**
    if *current_type est compatible avec les types de AllTypes[0]* **then**
        Ajouter current_type à AllType[0] et le retirer de AllTypes

// appel récursif pour faire un nouveau cluster avec les types restants
return ListCluster + Rec_Clusterize([], AllTypes)

Par la suite, dans chaque cluster les tâches seront triés de manières décroissantes, puis allouées par *Greedy_Cluster* avec la méthode de dichotomie décrite ci-dessus : L'allocation est testée pour un certain *epsilon* dans $[LB_{type}; 2 * LB * max_{coeff}]$ et l'intervalle diminué de moitié jusqu'à atteindre la précision voulue.

Algorithm 6: Greedy_Cluster / Complexité : 0(Ntypes*n*n + 3*n)

Entree: LM vide, LT_clusterised (LT ou les tâches sont regroupées par clusters) , epsilon
Sortie : LM allouée si epsilon convient, exception (epsilon trop petit) sinon

indMachineCourante = O **for** *cluster in LT_clusterised* **do**
    for *task in cluster* **do**
        if $indMachineCourante < len(LM)$ **then**
            Si l'ajout temporaire de task à MachineCourante ne fait pas dépasser son cout de epsilon, on l'ajoute pour de bon
            Sinon, on passe à la machine suivante si possible ; si impossible, epsilon trop petit
// Quand eps est trop petit on quitte ou on ajoute les taches restantes à la première machine
return *LM*

Dans une autre version de Greedy_Cluster, quand eps est trop petit il reste des tâches à allouer. Ces tâches seront alloués à l'aide de lpt-typed, permettant d'obtenir un meilleur eps. Les résultats sont presque semblable à lpt-typed puisque dans certains cas, il s'agit au final d'une allocation grâce à lpt-typed. Complexité similaire au précèdent mais la complexité de lpt-typed est à prendre en compte. Ce qui a pour effet de ralentir l'algorithme.

3.3.1 Order_Type : Classement des types

Order_Type est la version "continue" de Greedy-Cluster. En effet, on peut dire que ce dernier alloue les tâches d'un même cluster indifféremment de leur type, de façon "discrète" : les tâches de clusters différents ne peuvent se trouver sur une même machine. Order_Type, de son côté, établit un ordre des types, tel que les types côtes à côtes soient compatibles.

Nous avons testé plusieurs manières d'ordonner les types, dont :

- un ordonnancement glouton qui choisit le type suivant qui partage le coefficient le plus faible avec le type précédent Sa complexité est en $O(n_{types}^2)$
- un ordonnancement qui teste toutes les permutations de types et choisit celle qui minimise le produit des coefficients partagés par les types côte à côte. Son inconvénient est qu'il est exponentiel en le nombre de type, on ne l'utilise donc que lorqu'il y a peu de types.

Une fois les types ordonnés et les tâches classées par types, Order_Type alloue les tâches par la méthode de la dichotomie. L'allocation est testée pour un certain *epsilon* dans $[LB_{type}; 2 * LB * max_{coeff}]$ et l'intervalle diminué de moitié jusqu'à atteindre la précision voulue.

Algorithm 7: Order_Types

Entree: LT triée par type de OrderedType, epsilon
Sortie : LM allouée si epsilon convient, exception sinon

$indMachine = 0$
for *tache in LT* **do**
    $MachineCourante = LM[indMachine]$
    Ajouter tache à MachineCourante
    Retirer tache de LT
    if $CoutMax(MachineCourante, MCoeff) > epsilon * LB_{type}$ **then**
        Retirer tache de MachineCourante
        Remettre tache dans LT
        if *indMachine == len(LM)* **then**
            // plus de machine disponible
            Lever Exception : epsilon est trop petit
        else
            // passer à la machine suivante
            $indMachine$ += 1

// Si une machine est restée vide, on la remplit de la dernière tâche des machines les plus chargées tant que son CoutMax est inférieur au CoutMax de la machine la plus chargée

return LM

L'avantage d'Order_Type est sa rapidité : Il a une complexité en $O(n_{types}^2 * n^3 + m)$. Comme, Schedule_Juxtaposed, il se montre efficace dans les instances ou les tâches ne sont pas compatibles entre elles. Par contre, il sera moins efficace si les types sont compatibles, puisqu'il tend à disposer sur une même machine des tâches de même type dont le coefficient partagé est $1 > 0$.

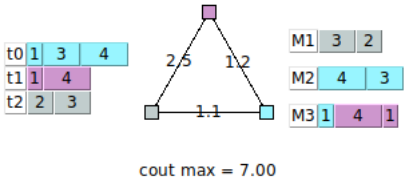


Figure 6 – tâches à allouer, matrice de types incompatibles, et allocation avec OrderType (glouton)

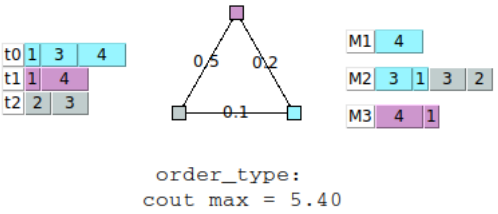


Figure 7 – tâches à allouer, matrice de types incompatibles, et allocation avec OrderType (glouton)

4 Résultats