École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

Département Informatique
5e année
2012 - 2013

Rapport PFE

Résolution d'un problème off-line de consolidation de serveurs

Encadrant
Vincent T'Kindt
vincent.tkindt@univ-tours.fr

Étudiant
Cyrille PICARD
cyrille.picard@etu.univ-tours.fr

Université François-Rabelais, Tours

DI5 2012 - 2013

Version du 26 mai 2013

Table des matières

Remerciement

Je tiens à remercier mon encadrant M. T'Kindt pour son aide et ses conseils tout long du projet. Ainsi que pour les éléments qu'il m'a fournis et qui m'ont été utiles pour la réalisation du projet et particulièrement la campagne de test.

Introduction

Dans le cadre du projet de fin d'étude, je me suis intéressé au sujet portant sur le problème de consolidation de serveur et plus particulièrement la partie appelé "oof-line". J'ai donc réalisé une heuristique de liste permettant d'obtenir une solution à ce problème plus rapidement que par l'utilisation de la méthode éxacte.

La première partie de ce rapport portera sur la présentation détaillé du problèmes ainsi que le modèle mathématique associé. La compréhension de ces deux points étaient très important pour partir dans le bonne direction pour la réalisation de l'heuristique.

La deuxième partie portera sur la mise en oeuvre de l'heuristique, en détaillant les principaux alogrithmes qui sont mise en utilisé avec des exemples d'applications. Il sera aussi aborté l'implémentation de cette heuristique ainsi que les résultats obtenus.

La dernière étape de ce rapport concerne le déroulement et la synthèse du projet.

1. Présentation du problème

Le problème de consolidation de serveur correspond au problème rencontré par les fournisseurs d'infrastructure (type Amazon, Google, ...) qui louent des ressources informatiques (CPU, GPU, Ram, disque) à des clients qui sont des fournisseurs de service. Ceux-ci achètent des machines virtuelles exécutées sur des ordinateurs mis en réseau.
L'objectif de ce problème est de déterminer un ordonnancement des machines virtuelles sur des machines physiques de façon à respecter les contraintes de ressources et à minimiser leurs coûts d'utilisation.
On distingue deux cas de figures au sein du problème de consolidation de serveur :

- Le premier est le cas "off-line" : Dans ce cas, on cherche à définir un ordonnancement pour les différents services à planifier sur les ressources physiques sur un horizon de planification sans tenir compte de l'évolution du réseau au sein du data center.
- Le deuxième est le cas "on-line" : Dans ce cas, on cherche toujours à définir l'ordonnancement comme dans le cas précédent, cependant on tient compte de toutes les modifications de réseau en temps réel et on adapte l'ordonnancement initialement prévu.

Le problème, suivant la description faite dans l'introduction, est de trouver un ordonnancement de tâches requises par des clients sur un ensemble de machines. Dans un premier temps nous allons définir les différents paramètres du problème à prendre en compte.

1.1 Notion de Machine virtuelle

Comme évoqué précédemment, les tâches sont des machines virtuelles qui sont à exécuter sur des machines physiques.

Chaque machine virtuelle est définie selon ses besoins en termes de ressources physiques qui sont définies de la façon suivante :

- n_s^c La charge nécessaire de CPU
- n_s^g La charge nécessaire de GPU
- n_s^r La quantité de RAM requise
- n_s^h La quantité de mémoire requise sur le disque dur

Dans notre problème, on s'intéresse à l'ordonnancement des tâches sur des ressources. On considére alors les machines virtuelles comme des travaux à assigner sur des machines.

On définit pour chaque machine virtuelle i le fait qu'elle doit être exécutée ou non à un instant de temps t à l'aide de $u_{i,t}$. Si la tâche i doit être exécutée à l'instant de temps t alors $u_{i,t} = 1$ sinon $u_{i,t} = 0$.

On introduit la notion d'affinité entre deux machines virtuelles correspondant au fait que si l'exécution d'une tâche i est fortement liée à l'exécution d'une tâche j alors il y a des échanges de flots de données entre les deux machines virtuelles correspondantes. Cela implique que si l'une des deux machines n'est pas en route, l'autre machine ne peut s'exécuter. On définit cette affinité avec une variable booléenne de cette façon $a_{i,k} = 1$ si il existe une affinité entre les travaux i et k et on définit

l'ensemble de toutes les affinités par $A = (a_{i,k})$ la matrice des affinités.

De plus, certaines machines virtuelles peuvent être migrées par souci d'optimisation de l'utilisation du réseau ou du nombre de machines physiques allumées. Ces migrations sont considérées comme des migrations lives, c'est à dire que la machine virtuelle qui est en cours de migration continue de s'exécuter sur la machine source pendant que les données sont transférées, après quoi la tâche est éteinte sur la machine d'origine pour être démarrée sur la machine de destination.

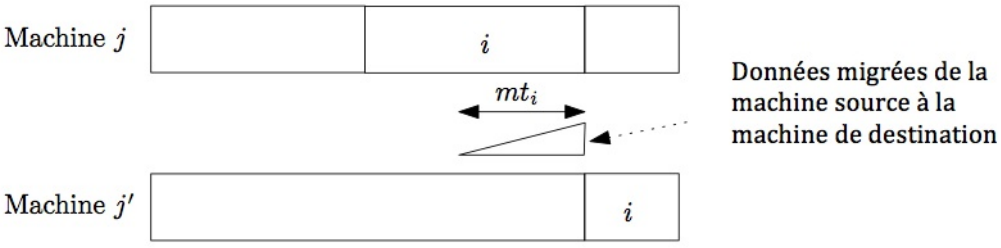

FIGURE 1.1 – Représentation de l'opération de migration

Parmi les machines virtuelles, certaines sont préemptables, c'est à dire qu' elles peuvent ne pas être exécutées même si elles sont sensées l'être d'après leurs $u_{i,t}$.
Dans ce cas, si une tâche est suspendue à un instant t, elle ne peut pas être résumée avant un certain temps qui correspond au temps nécessaire pour charger en mémoire le contexte d'exécution. Chaque machine physique possède une vitesse v_j de chargement du contexte, on définit alors que le temps de resume de la tâche i sur la machine j est défini de cette façon : $rt_{i,j} = \frac{n_j^r}{v_j}$.
On peut donc dire que le temps total de préemption est au moins égale à $rt_{i,j}$

FIGURE 1.2 – Représentation de l'opération de suspend et de resume

1.2 Notion de machine physique et réseau

1.2.1 Réseau

Les machines physiques chez le fournisseur d'infrastructure sont réparties dans un centre appelé data center. Cependant, il faut pour la suite du problème définir une architecture du réseau permettant de relier l'ensemble des machines du data center.

La structure choisie pour représenter au mieux la structure du réseau au sein du data center est une structure sous la forme d'un arbre. Au sein de cet arbre, le nœud racine représente le point d'entrée du réseau, les nœuds suivants correspondent au switch à l'intérieur du réseau et les feuilles représentent les machines physiques. La figure 1.3 est un exemple de cette représentation sous forme d'arbre.

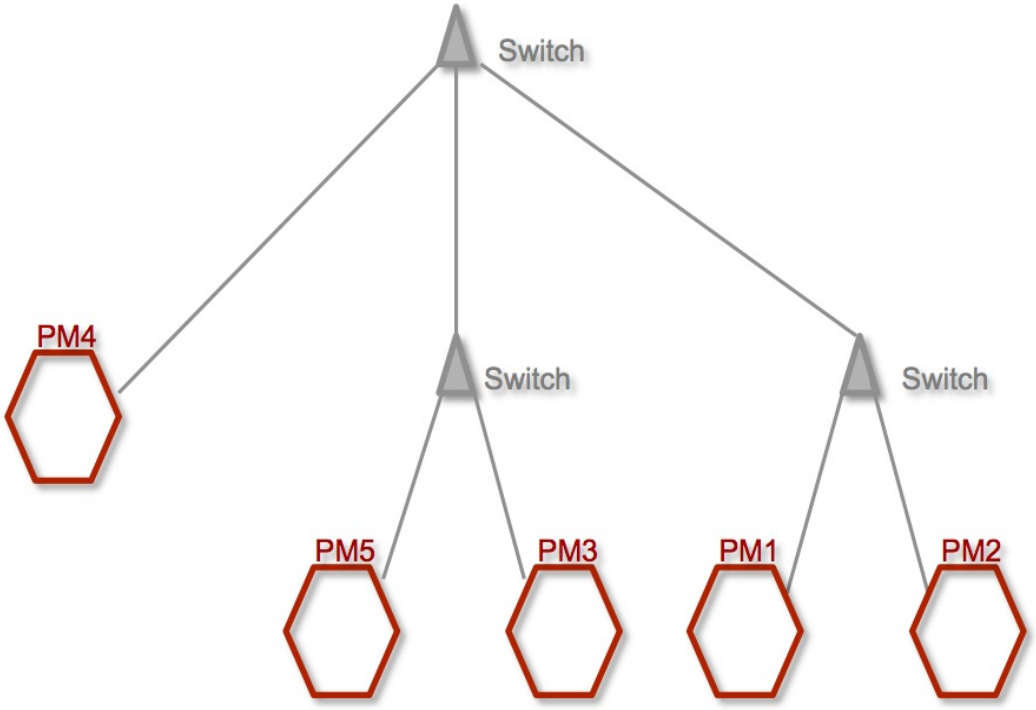

FIGURE 1.3 – Représentation du réseau sous forme d'arbre

On suppose alors que le réseau situé chez le fournisseur possède une taille relativement peu étendue afin de considérer que le temps de transfert sur le réseau est négligeable et que tout les liens sont équivalent en termes de capacité de transport de l'information. On définit par ailleurs aussi $\mathcal{P}$ qui représente l'ensemble des chemins $P_{j,j'}$ entre deux machines j et j', chaque chemin représente une succession de liens utilisé pour passer du sommet correspondant à la machine j au sommet correspondant à la machine j'. Chaque lien peut alors appartenir à plusieurs chemins différents c'est pourquoi b représente le maximum de bande passante pouvant être utilisé sur n'importe quel chemin $P_{j,j'}$.

1.2.2 Machine physique

Chaque machine physique chez l'hébergeur est représentée par ses caractéristiques, c'est à dire les ressources physiques dont elle dispose pour exécuter les différents services. Parmi les ressources disponibles on trouve celles qui concernent le CPU, celles concernant la GPU ainsi que celles concernant la mémoire (RAM, Disque dur). On définit alors une machine physique j de la manière suivante, pour tout instant t on a :

- m_j^c la charge maximale acceptée par le CPU,
- m_j^g la charge maximale acceptée par le GPU,
- m_j^r la capacité (en Mb) de RAM disponible,
- m_j^h la capacité (en Mb) d'espace mémoire disponible sur les disques durs.

De plus, on définit pour une machine physique j les coûts d'utilisation de la manière suivante :

- α_j^c le coût d'utilisation CPU,
- α_j^g le coût d'utilisation GPU,
- α_j^r le coût d'utilisation d'un Mb de RAM,

– α_j^h le coût d'utilisation d'un Mb de disque dur.

On ajoute aussi le coût d'utilisation intrinsèque de la machine à l'instant t que l'on notera β_t. Ce coût d'utilisation intrinsèque correspond au fait qu'à partir du moment où une machine physique est mise en marche, elle possède une consommation électrique qui ne dépend pas de l'exécution de la machine virtuelle, cette consommation représente un coût dont il faut tenir compte.

L'objectif du problème de consolidation de serveur étant de minimiser le coût total d'utilisation des machines.

2. Modèle mathématique

Maintenant que le problème a été présenté, nous allons nous intéresser au modèle mathématique qui va être mis en place pour résoudre le problème et trouver un ordonnancement. Le modèle mathématique qui va être utilisé est un modèle "time indexed formulation".

Voici dans un premier temps un tableau récapitulatif des différents paramètres du problème défini précédemment.

Données générales du problème :	
T,	horizon de planification,
N,	le nombre de tâches,
M,	le nombre de machines physiques,
Pour les tâches :	
n_i^c	la charge requise par la tâche i en termes de CPU,
n_i^g	la charge requise par la tâche i in termes de GPU,
n_i^r,	la quantité de RAM requise par la tâche i,
n_i^h,	la capacité de disque dur requise par la tâche i,
$u_{i,t}$,	un booléen indiquant si la tâche i est susceptible d'être traitée au temps t,
$A = (a_{i,k})_{i,k}$,	matrice des affinités,
$Q = (q_{i,j})_{i,j}$,	matrice des pré-affectation,
$b_{i,j}$,	la bande passante requise par les tâches i et j pour communiquer,
R_i,	un booléen indiquant si la tâche i est préemptable ou non,
ρ_i,	a pénalité unitaire par tâche suspendu i, si préemptable ($R_i = 1$), 0 sinon,
mt_i	le temps de migration de la tâche i, avec $mt_i = \lceil \frac{n_i^h + n_i^r}{b_{i,i}} \rceil$,
$rt_{i,j}$	le temps de reprise i sur la machine j, avec $rt_i = \lceil \frac{n_i^r}{v_j} \rceil$,
Pour les machines physiques :	
m_j^c	la charge maximale acceptée par CPUs de la machine j,
m_j^g	la charge maximale acceptée par GPUs de la machine j,
m_j^r,	la quantité de RAM de la machine j,
m_j^h,	la capacité du disque dur de la machine j,
m_j^b,	le maximum de bande passante qui peut être utilisé par n'importe quelle tâche sur cette machine,
α_j^c,	le coût d'utilisation du CPU de la machine j,
α_j^g,	le coût d'utilisation du GPU de la machine j,
α_j^r,	le coût d'utilisation d'un Mb de RAM de la machine j,
α_j^h,	le coût d'utilisation d'un Mb de capacité disque de la machine j,
β_t,	le coût d'une machine en marche à l'instant t,
v_j,	la rapidité de la machine j pour charger le contexte d'exploitation,
Pour le réseau :	
$G = (V, E)$,	le graphe modélisant le réseau,
b,	la bande passante maximale associée pour toutes les arrêtes $e_{\ell,\ell'} \in E$,
$\mathcal{P}$,	un ensemble de chemins entre deux machines, un chemin est un ensemble d'arrêtes,
$P_{\ell,\ell'}$,	l'ensemble des couples de machine (j, j') qui utilise l'arrête $e_{\ell,\ell'}$,

Table 2.1 – Notation permettant de définir les données du problème

2.1 Variables de décision

La "time indexed formulation" liée à ce problème est définie par les variables de décisions booléennes $x_{i,t}^{j}$:

$$x_{i,t}^{j} = \begin{cases} 1 & \text{si la tâche } i \text{ est traitée sur la machine } j \text{ dans l'interval } [t; t+1[, \\ 0 & \text{otherwise} \end{cases}$$

On définit aussi un ensemble de variables supplémentaires suivantes :

- $y_{i,i',t}^{j,j'} = 1$ si les tâches i et i' sont exécutées, respectivement, sur les machines j et j' à l'instant t et ont besoin de communiquer sur le réseau ; 0 dans le cas contraire. On spécifie que dans le cas où une tâche i migre à l'instant t depuis la machine j vers la machine j' alors $y_{i,i',t}^{j,j'}$ vaut 1 ; cela implique que nous avons toujours $y_{i,i',t}^{j,j'} = 0$.
- $z_{t,j} = 1$ si la machine j traite au moins une tâche à l'instant t ; 0 sinon.
- z_t est le nombre de machines qui sont en marche à l'instant t, *i.e.* qui exécutent au moins une tâche.
- $d_{i,t}$ est la durée des opérations de reconfiguration des tâches i à l'instant t : c'est aussi la durée de l'opération de "resume" commençant à t ou la durée de l'opération de migration finissant à l'instant t.

2.2 Modélisation du Problème de Consolidation de serveur

On peut modéliser le problème de consolidation de serveur de la manière suivante :

Minimiser $TC = \sum_{t=1}^{T}\sum_{i=1}^{N}\sum_{j=1}^{M}\left(x_{i,t}^{j}(\alpha_j^c n_i^c + \alpha_j^g n_i^g + \alpha_j^h n_i^h + \alpha_j^r n_i^r)\right) +$
$\sum_{t=1}^{T}\sum_{i=1}^{N}(1 - \sum_{j=1}^{M} x_{i,t}^{j})\rho_i u_{i,t} + \sum_{t=1}^{T}\beta_t z_t$
et
Minimiser $RE = \sum_{i=1}^{N}\sum_{t=1}^{T} d_{i,t}$
sous contraintes

$$\sum_{i=1}^{N} n_i^c x_{i,t}^{j} \leq m_j^c \qquad \forall t = 1,...,T, \forall j = 1,...,M \quad \text{(A)}$$

$$\sum_{i=1}^{N} n_i^g x_{i,t}^{j} \leq m_j^g \qquad \forall t = 1,...,T, \forall j = 1,...,M \quad \text{(B)}$$

$$\sum_{i=1}^{N} n_i^h x_{i,t}^{j} + \sum_{k=1,k\neq j}^{M} n_i^h y_{i,i,t}^{k,j} \leq m_j^h \qquad \forall t = 1,...,T, \forall j = 1,...,M \quad \text{(C)}$$

$$\sum_{i=1}^{N} n_i^r x_{i,t}^j + \sum_{k=1,k\neq j}^{M} n_i^r y_{i,i,t}^{k,j} \leq m_j^r \qquad \forall t = 1,...,T, \forall j = 1,...,M \qquad \text{(D)}$$

$$y_{i,i',t}^{j,j'} \geq x_{i,t}^j + x_{i',t}^{j'} - 1 \qquad \forall t = 1,...,T,\ \forall i,i' = 1,...,N,\ i \neq i' \text{ tel que } a_{i,i'} = 1,\ \forall j,j' = 1,...,M, j \neq j' \qquad \text{(E)}$$

$$y_{i,i,t}^{j,j'} \geq (x_{i,t_1-1}^j + x_{i,t_1}^{j'} - 1) \qquad \forall i = 1,...,N,\ \forall j,j' = 1,...,M, j \neq j',\ \forall t = mt_i,...,T,\ \forall t_1 = t+1,...,t+mt_i \qquad \text{(F)}$$

$$\sum_{j=1}^{M} x_{i,t}^j = u_{i,t} \qquad \forall i = 1,...,N, \text{ tel que } R_i = 0, \forall t = 1,...,T, \qquad \text{(G)}$$

$$\sum_{j=1}^{M} x_{i,t}^j \leq u_{i,t} \qquad \forall i = 1,...,N, \text{ tel que } R_i = 1, \forall t = 1,...,T, \qquad \text{(H)}$$

$$x_{i,t}^j \leq u_{i,t}(1 - q_{i,k}) \qquad \forall t = 1,...,T,\ \forall j,k = 1,...,M,\ k \neq j,\ \forall i = 1,...,N, \qquad \text{(I)}$$

$$\sum_{(j,j')\in P_{\ell,\ell'}} \sum_{i,i'=1,a_{i,i'}=1}^{N} b_{i,i'} y_{i,i',t}^{j,j'} \leq b \qquad \forall t = 1,...,T,\ \forall e_{\ell,\ell'} \in E \qquad \text{(J)}$$

$$z_{t,j} \geq x_{i,t}^j \qquad \forall t = 1,...,T, \forall i = 1,...,N, \forall j = 1,...,M \qquad \text{(K)}$$

$$z_t = \sum_{j=1}^{M} z_{t,j} \qquad \forall t = 1,...,T \qquad \text{(L)}$$

$$x_{i,t_2}^j \leq 1 - x_{i,t_1}^j + x_{i,t_1+1}^j \qquad \forall t_1 = 1,...,(T - rt_{i,j}),\ \forall t_2 = t_1+1,...,t_1 + rt_{i,j}, \forall j = 1,...,M, \forall i = 1,...,N \text{ tel que } R_i = 1 \qquad \text{(M)}$$

$$d_{i,t_1} \geq (t_2 - t_1)(x^j_{i,t_1-1} - \sum_{k=1}^{M} \sum_{t=t_1+1}^{t_2-1} x^k_{i,t} - 1 + x^j_{i,t_2}) \quad \forall t_1 = 1, ..., (T - rt_{i,j} - 1),\ \forall t_2 = t_1 + rt_{i,j} + 1, ..., T,\ \forall j = 1, ..., M,\ \forall i = 1, ..., N \text{ tel que } R_i = 1 \quad \text{(N)}$$

$$d_{i,t_1} \geq mt_i(x^j_{i,t_1} + x^{j'}_{i,t_1+1} - 1) \quad \forall t_1 = 1, ..., (T - mt_i) - 1),\ t_2 = t_1 + mt_i + 1,\ \forall j, k = 1, ..., M,\ j \neq k,\ \forall i = 1, ..., N \quad \text{(O)}$$

$x^j_{i,t} \in \{0; 1\}$, $y^{j,j'}_{i,i',t} \in \{0; 1\}$, $z_{t,j} \in \{0; 1\}$, $z_t \geq 0$, $d_{i,t} \geq 0$

Les contraintes (A) garantissent que pour la machine physique j, la somme des besoins en CPU des tâches affectées à la machine j ne dépasse pas la capacité totale de CPUs de la machine j tandis que les contraintes (B) définissent de la même manière l'utilisation de la GPUs.
Les contraintes (C) et (D) complètent les contraintes d'intégrités concernant les ressources physiques : contraintes (C) (resp. (D)) implique que sur chaque machine les tâches exécutées ou migrées sur la machine ne puissent utilisées plus que l'espace disque disponible (resp. RAM).
Les contraintes (E) et (F) définissent l'utilisation du réseau, en termes de communication et de migrations, à n'importe quel instant t. D'après les contraintes (F) nous pouvons déduire qu'à l'instant t un job i est en train de migrer depuis la machine j vers la machine j' si il existe une fenêtre de temps telle que $t \in [t_1 - mt_i; t_1]$ de longueur mt_i telle que la migration commence à l'instant $(t_1 - mt_i)$ et la tâche i change de machine à l'instant t_1.
(Figure 2.1).

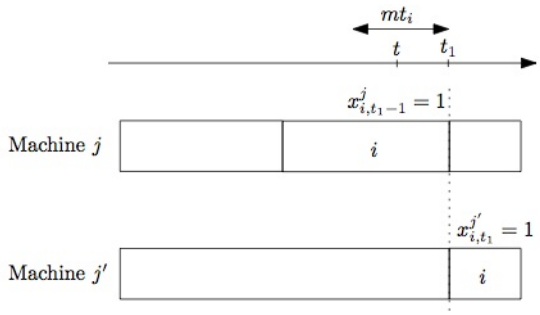

FIGURE 2.1 – Illustration des contraintes F

Les contraintes (G) impliquent qu'à chaque instant de temps t précis et pour chaque machine virtuelle non préemptable qui doit être exécutée sur une machine physique même si elle est entrain d'être migrée à travers le réseau. Les contraintes (H) imposent qu'une machine virtuelle préemptable peut être interrompue même si elle est planifiée pour être exécutée. Les contraintes (I) correspondent à des contraintes de pré-affectation des tâches. Alors que les contraintes (J) définissent les contraintes de capacités de la bande passante qui peuvent être utilisées aussi bien pour la communication entre deux tâches que pour la migration.
Les contraintes (K) et (L) définissent le nombre de machines qui peuvent être en marche à l'instant t. Les contraintes (M) définissent le temps minimum avant lequel une tâche suspendue peut être résumée. D'après ces contraintes ; on peut déduire qu'entre le moment $(t_1 + 1)$ où l'opération de "suspend" est réalisée et l'instant t_2 où celle de "resume" est réalisée, une tâche préemptable job i sur la machine j ne doit pas être exécutée et nous devons avoir au moins $(t_2 - t_1) = rt_{i,j} + 1$ avant de pouvoir réaliser l'opération de "resume".
(Figure 2.2). Les contraintes (N) définissent les durées des opérations de "resume" commencées à

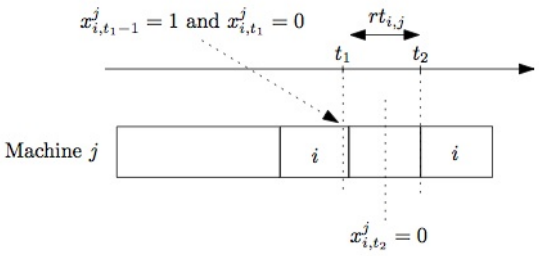

FIGURE 2.2 – Illustration des contraintes M

l'instant t_1 et les contraintes (O) correspondent aux durées des opérations de "migration" complètes à l'instant $(t_1 + 1)$ pour toutes les tâches i.

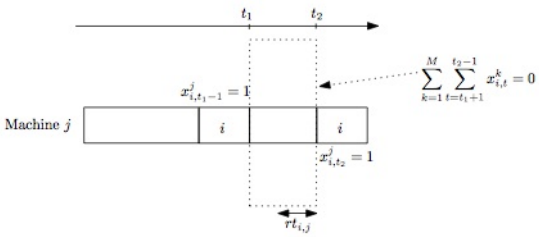

FIGURE 2.3 – Illustration des contraintes N

3. Résolution du problème

3.1 Description de l'heuristique

L'idée pour résoudre ce problème qui est NP-complet au sens fort de manière pseudo polynomiale est d'utiliser une heuristique de listes. Pour ce faire, on définit chaque machine physique comme un bin-packing qui va être rempli par des machines virtuelles.
Cependant il faut bien comprendre que les bin-packings utilisés ne sont pas des bin packing à quatre dimensions où chacune représente une caractéristique physique tel le CPU, GPU, RAM, HDD, ce sont des bin-packing contraints à quatre dimensions.
On entend par bin-packing contraint le fait qu'une fois qu'un objet est placé à l'interieur du bin, les ressources représentées par les dimensions du bin utilisé par ce dernier ne peut pas être partagé avec un autre objet que l'on voudrait aussi mettre dans le bin.
Dans notre cas une machine virtuelle une fois affectée à un serveur (bin), cette dernière utilise une partie des ressources GPU,CPU,RAM et HDD du bin, cette partie des ressources n'est alors plus disponible pour affecter une nouvelle machine sur le serveur. L'image 3.1 est une représentation de cette modélisation à l'aide de bin packing contraint.

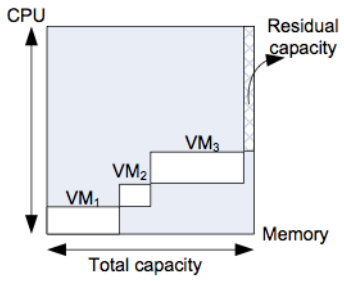

FIGURE 3.1 – Représentation de la modélisation sous forme de bin-packing contraint

En réalisant cette modélisation sous forme de bin-packing, l'idée est de diviser l'horizon de planification en intervalles sur lesquels les $u_{i,t}$ sont constants afin de pouvoir faire une affectation des tâches qui doivent être exécutées sur chaque intervalle. De plus, comme on essaye de minimiser le coût d'utilisation, on va ordonner les serveurs par ordre croissant de leur coût d'utilisation normalisé respectif, afin de commencer par remplir les serveurs qui sont les plus intéressants à remplir pour la fonction objectif.

Une fois qu'on a notre liste de serveurs classés et notre liste d'intervalles, on réalise les opérations suivantes afin de remplir les bins avec les machines virtuelles. Sur chaque intervalle on crée les listes des tâches qui doivent être affectées, on calcule les priorités de chaque tâches pour classer les listes par ordre décroissant de priorité puis on affecte les tâches sur les machines.

3.2 Algorithmes

Dans cette partie, les différents algorithmes permettant de réaliser les opérations décrites si dessus, afin d'obtenir notre ordonnancement des machines virtuelles sur les machines physiques, vont être décrits et seront suivis d'exemples d'application sur un petit jeu de données.

Pour simplifier l'écriture des algorithmes, les données sont contenues dans une structure de données appelée DATA et un certain nombre de listes sont aussi contenues dans une structure de données appelée TRAIT.

Je vais commencer par énumérer un certain nombre de tableaux contenus dans la structure de données qui seront utilisés dans les algorithmes qui suivent.

- ListeIntervalles[HorizonPlanification] permet de stocker pour chaque intervalle calculé la borne inférieure et supérieure de l'intervalle.
- ListeServeur[NombreMachines] permet de stocker l'indice de la machine physique, ses caractéristiques au niveau des ressources, ainsi que son coût d'utilisation normalisé.
- ListOrdo[HorizonPlanification][NombreTaches] permet de stocker pour chacune des tâches l'indice de la machine physique sur laquelle a été affecté à chaque instant de temps. Sachant que si la tâche n'est pas sensée s'exécuter à un instant l'indice de la machine sera -2, et si elle n'est pas affectée alors qu'elle doit s'exécuter l'indice de la machine prendre la valeur -1.
- ListeServeurON[NombreMachines] permet de stocker l'indice des serveurs qui est allumé

Pour faciliter la comphréension des différents algorithmes, chaque alogirthme sera déroulé sur les données suivante :

```
12 //horizon de planification
6 //nombre de machines virtuelles
3 //nombre de machines physiques
//détails des besoins des machines virtuelles sous la forme CPU GPU HDD RAM Ri Vj
40 10 80 100 0 0
40 10 80 100 0 0
40 10 80 100 0 0
40 00 120 100 0 0
40 10 80 100 1 1000000
60 00 150 20 1 5
//matrice des uit
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 0 0 1 1 1 1 1 1
1 1 0 0 0 0 0 1 1 1 1 0
1 1 0 0 0 0 0 1 1 1 1 0
1 1 0 0 0 0 0 1 1 1 1 0
1 1 0 0 0 0 0 1 1 1 1 0
//matrice des affinités
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 1 0 0
0 0 1 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
//matrice de préaffectation
1 0 0
1 1 1
```

```
1 1 0
1 1 1
1 1 1
1 1 1
//Caractéristique des machines physiques
250 150 1000 250 10
300 220 3000 400 12
250 150 1000 250 10
//cout d'utilisation pour chaque ressources
2 4 1 4
1 2 7 3
1 1 1 1
//cout unitaire pour chaque instant de temps
1 1 1 2 2 2 1 1 1 3 3 3
100
10
10 20 20 20 20 20
20 10 10 10 10 20
20 10 10 10 10 10
20 10 10 10 20 10
20 10 10 10 20 10
10 10 10 10 10 10
8
3 1 2 1 3 2 3
3 1 2 1 3 2 3
2 1 2 1 3
3 1 2 1 3 2 3
2 1 2 1 3
2 2 3 1 3
2 1 2 1 3
2 2 3 1 3
```

3.2.1 Algorithme de calcul des intervalles

Le premier algorithme 1 qui rentre en compte dans l'heuristique est celui qui permet de générer la liste des intervalles tel que sur chaque intervalle les $u_{i,t}$ restent constant.

Algorithme 1 Calcul des Intervalles

Précondition:
Les données concernant la planification sont contenues dans la structure de données Data

Postcondition:
Sortie : tableau contenant l'ensemble des intervalles

```
1: int nombreIntervalle = 0 ;
2: int instant = 0 ;
3: int tacheI = 0 ;
4: int indice = 0 ;
5: /* T() permet d'accéder à la donnée horizon de planification et N() permet d'accéder à la donnée nombre de tache */
6: /* La fonction u(i,t) permet d'accéder à la données u_{i,t} de la tache i à l'instant t */
7: pour instant allant de 0 à T()-1 faire
8:     pour tacheI allant de 0 à N()-2 faire
9:         si (u(tacheI,instant)=u(tacheI,instant+1))&&(u(tacheI+1,instant)≠u(tacheI+1,instant+1)) alors
10:            si indice = 0 alors
11:                ListeIntervalles[indice].BorneInf = 0 ;
12:                ListeIntervalles[indice].BorneSup = instant ;
13:                indice = indice + 1 ;
14:                nombreIntervalle = nombreIntervalle + 1 ;
15:            sinon
16:                ListeIntervalles[indice].BorneInf = ListeIntervalles[indice - 1].BorneSup+1 ;
17:                ListeIntervalles[indice].BorneSup = instant ;
18:                si ListeIntervalles[indice].BorneInf > ListeIntervalles[indice].BorneSup alors
19:                    ListeIntervalles[indice].BorneInf = 0 ;
20:                    ListeIntervalles[indice].BorneSup = 0 ;
21:                sinon
22:                    indice = indice + 1 ;
23:                    nombreIntervalle = nombreIntervalle + 1 ;
24:                fin si
25:            fin si
26:        fin si
27:    fin pour
28: fin pour
29: /* on vérifie que le dernier intervalle contient le dernier instant de planification et on l'ajoute si besoin */
30: si ListeIntervalles[indice - 1].BorneSup = T() - 1 alors
31:     indice = indice - 1 ;
32: sinon
33:     ListeIntervalles[indice].BorneInf = ListeIntervalles[indice - 1].BorneSup + 1 ;
34:     ListeIntervalles[indice].BorneSup = T() - 1 ;
35:     nombreIntervalle = nombreIntervalle + 1 ;
36: fin si
```

Cet algorithme donne le résultat suivant à partir des données précédement citées :

IntervalleNumero	0	1	2	3	4	5
BorneInf	0	2	4	6	7	11
BorneSup	1	3	5	6	10	11

3.2.2 Algorithmes de calcul des coûts

Une fois le calcul de l'ensemble des intervalles sur lesquels on va travailler, on cherche à calculer différents coûts qui vont servir pour l'exécution de l'heuristique. Je vais dans cette partie, présenter l'algorithme permettant de calculer les couts normalisés de chaque serveur de la liste des serveurs et de classer cette liste par coût normalisé croissant. 2. Ainsi que l'algorithme de calcul de la valeur de la fonction 4 et celui de calcul du cout d'affectation d'une tâche i sur une machine j 3.

Algorithme 2 Calcul Cout Normalisé Serveur

Précondition:

Les données concernant les caractéristiques des serveurs disponibles sont contenues dans la structure DATA et accessible à l'aide de fonction.

Postcondition:

Sortie : tableau des couts normalisés des serveurs ordonné par ordre croissant.

```
1: entier indiceServeur ;
2: entier j = 0 ;
3: réel CoutTotal ;
4: réel CoutNorm ;
5: réel SommeCaract ;
6: /* M() permet d'accéder au nombre de serveur */
7: pour indiceServeur allant de 0 à M()-1 faire
8:         ListeServeurs[j].IndiceServeur = indiceServeur ;
9:         CoutTotal = mc(indiceServeur)*alphac(indiceServeur) +
           mg(indiceServeur)*alphag(indiceServeur) + mr(indiceServeur)*alphar(indiceServeur)
           + mh(indiceServeur)*alphah(indiceServeur) ;
10:        SommeCaract = mc(indiceServeur)+mg(indiceServeur)+mr(indiceServeur)+mh(indiceServeur) ;

11:        CoutNorm = CoutTotal / SommeCaract ;
12:        ListeServeur[j].CoutNorm = CoutNorm ;
13:        j+1 ;
14: fin pour
15: Trie(ListeServeur) ;
```

En appliquant cet algorithme sur le jeux de données, on obtient le déroulement et les résultats suivants :

<u>déroulement de la boucle :</u>

indiceServeur = 0
ListeServeur[0].IndiceServeur = 0
CoutTotal = 250*2 + 150*4 + 1000*1 + 250*4 = 3100
SommeCaract = 250 + 150 + 1000 + 250 = 1650
CoutNorm = 3100/1650 = 1,88
ListeServeur[0].CoutNorm = 1,88

j = 1

indiceServeur = 1
ListeServeur[1].IndiceServeur = 1
CoutTotal = 300*1 + 220*2 + 3000*7 + 400*3 = 22940
SommeCaract = 300 + 220 + 3000 + 400 = 3920
CoutNorm = 22940/3920 = 5,85 ListeServeur[1].CoutNorm = 5,85

indiceServeur = 2
ListeServeur[2].IndiceServeur = 2
CoutTotal = 250*1 + 150*1 + 1000*1 + 250*1 = 1650
SommeCaract = 250 + 150 + 1000 + 250 = 1650
CoutNorm = 1650/1650 = 1 ListeServeur[2].CoutNorm = 1,00

Après la boucle on appel la fonction de trie qui classe par cout normalisé croissant la liste, on obtient alors le tableau suivant :

Indice Serveur	2	0	1
Cout Normalisé	1,00	1,88	5,85

Algorithme 3 Calcul Cout Affectation

Précondition:

Entrée : indice i de la machine virtuelle à affecter et indice j de la machine physique
Les données concernant les caractéristiques des serveurs disponibles sont contenues dans la structure DATA et accessible à l'aide de fonction.

Postcondition:

Sortie : la valeur du coût d'affectation de la machine physique i sur la machine physique j.

```
1: réel cout ;
2: réel coutCPU = alphac(j)*nc(i) ;
3: réel coutGPU = alphag(j)*ng(i) ;
4: réel coutRAM = alphar(j)*nr(i) ;
5: réel coutHDD = alphah(j)*nh(i) ;
6: cout = coutGPU + coutCPU + coutRAM + coutHDD ;
7: retourner cout ;
```

Pour illustrer cet algorithme, on va calculer le cout d'affectation de la deuxième machine virtuelle d'indice 1 sur la première machine physique de la liste des serveurs issue de l'utilisation de l'algorithme précédent. Cette affectation est possible d'après la matrice de préaffectation.

coutCPU = alphac(2) × nc(1) = 1 × 40 = 40 coutGPU = alphag(2) × ng(1) = 1 × 10 = 10 coutRAM = alphar(2) × nr(1) = 1 × 100 = 100 coutHDD = alphah(2) × nh(1) = 1 × 80 = 80 cout = 40 + 10 + 100 + 80 = 230

On obtient donc 230 pour le coût d'affectation de la tâche 1 sur la machine 2.

Algorithme 4 calculCoutTotal

Précondition:

Entrée : ListOrdo qui correspond à la matrice d'affectation des machines virtuelles sur les machines physique

Les données concernant les caractéristiques des serveurs disponibles sont contenues dans la structure DATA et accessible à l'aide de fonction.

Postcondition:

Sortie : la valeur du coût total de l'ordonnancement des VMs sur les serveurs

```
1: entier TotalCost = 0 ;
2: entier CoutGPU = 0 ;
3: entier CoutCPU = 0 ;
4: entier CoutRAM = 0 ;
5: entier CoutHDD = 0 ;
6: entier CoutUnitaire = 0 ;
7: entier Penalite = 0 ;
8: /* M() permet d'accéder au nombre de serveur, N() permet d'accéder au nombre de tache et T()
   permet d'accéder à la valeur de l'horizon de planification */
9: pour t allant de 0 à T() faire
10:     pour n allant de 0 à N() faire
11:         si (ListOrdo[t][n].IndiceMachine ≠ -1)&&(ListOrdo[t][n].IndiceMachine ≠ -2) alors
12:             CoutGPU = CoutGPU + alphag(ListOrdo[t][n].IndiceMachine)*ng(n) ;
13:             CoutCPU = CoutCPU + alphac(ListOrdo[t][n].IndiceMachine)*nc(n) ;
14:             CoutRAM = CoutRAM + alphar(ListOrdo[t][n].IndiceMachine)*nr(n) ;
15:             CoutHDD = CoutHDD + alphah(ListOrdo[t][n].IndiceMachine)*nh(n) ;
16:         fin si
17:         si ListOrdo[t][n].IndiceMachine = -1 alors
18:             Penalite = Penalite + rho(n) ;
19:         fin si
20:     fin pour
21:     pour z_t allant de 0 à NbServeurON faire
22:         CoutUnitaire = CoutUnitaire + z_t*beta(t) ;
23:     fin pour
24: fin pour
25: TotalCost = CoutGPU + CoutCPU + CoutRAM + CoutHDD + CoutUnitaire + Penalite ;
26: retourner TotalCost ;
```

3.2.3 Algorithme de construction des listes

Dans cette partie je vais présenter les algorithmes qui permettent de construire les listes des machines virtuelles à affecter. Le premier algorithme présenté sera celui qui permet de créer les quatres listes des tâches non préemptables qui doivent être exécutées sur l'intervalle étudié. Les quatres listes sont :

- ListeGPU1 : liste des VMs ayant des besoins en GPU et CPU et tel que besoins HDD > besoins RAM.
- ListeGPU2 : liste des VMs ayant des besoins en GPU et CPU et tel que besoins RAM > besoins HDD.
- ListeCPU1 : liste des VMs ayant des besoins uniquement en CPU et tel que besoins HDD > besoins RAM.

– ListeCPU2 : liste des VMs ayant des besoins uniquement en CPU et tel que besoins RAM > besoins HDD.

Les autres algorithmes permettent de construire les listes pour les tâches préemptables.

Algorithme 5 Création des listes tâches non préemptables

Précondition:

Entrée : indice de l'intervalle sur lequel on travaille.

Postcondition:

Sortie : listes des machines virtuelles à affecter sur l'intervalle, on a alors 4 listes afin d'avoir deux liste pour les VMs qui ont besoins de GPU et deux autres pour les VMs qui n'ont pas besoins de GPU

```
1:  pour temps allant de ListeIntervalles[indice].BorneInf à ListeIntervalles[indice].BorneSup faire
2:      entier indiceListeGPU1 = 0 ;
3:      entier indiceListeGPU2 = 0 ;
4:      entier indiceListeCPU1 = 0 ;
5:      entier indiceListeCPU2 = 0 ;
6:      entier NombreTachesListeGPU1 = 0 ;
7:      entier NombreTachesListeGPU2 = 0 ;
8:      entier NombreTachesListeCPU1 = 0 ;
9:      entier NombreTachesListeCPU2 = 0 ;
10:     pour tache allant de 0 à N() faire
11:         si u(tache,temps)=1 && ng(tache)>0 && R(tache)=0 alors
12:             si nh(tache)>nr(tache) alors
13:                 ListeGPU1[indiceListeGPU1].IndiceVM = tache ;
14:                 indiceListeGPU1 = indiceListeGPU1 + 1 ;
15:                 NombreTachesListeGPU1 = NombreTachesListeGPU1 + 1 ;
16:             sinon
17:                 ListeGPU2[indiceListeGPU2].IndiceVM = tache ;
18:                 indiceListeGPU2 = indiceListeGPU2 + 1 ;
19:                 NombreTachesListeGPU2 = NombreTachesListeGPU2 + 1 ;
20:             fin si
21:         sinon
22:             si u(tache,temps)=1 && R(tache)=0 alors
23:                 si nh(tache)>nr(tache) alors
24:                     ListeCPU1[indiceListeCPU1].IndiceVM = tache ;
25:                     indiceListeCPU1 = indiceListeCPU1 + 1 ;
26:                     NombreTachesListeCPU1 = NombreTachesListeCPU1 + 1 ;
27:                 sinon
28:                     ListeCPU2[indiceListeCPU2].IndiceVM = tache ;
29:                     indiceListeCPU2 = indiceListeCPU2 + 1 ;
30:                     NombreTachesListeCPU2 = NombreTachesListeCPU2 + 1 ;
31:                 fin si
32:             fin si
33:         fin si
34:     fin pour
35: fin pour
```

Si on exécute cet algorithme sur le premier intervalle, son déroulement est le suivant :

indice = 0 ;
temps = ListeIntervalles[indice].BorneInf = 0 ;

tache = 0 ;
u(tache,temps)=1 & ng(tache)=40 > 0 & R(tache)=0 ;
nh(tache)=80 & nr(tache) = 100 ;
nh(tache)<nr(tache) ;
ListeGPU2[indiceListeGPU2].IndiceVM = 0 ;
indiceListeGPU2 = 1 ;
NombreTachesListeGPU2 = 1 ;

tache = 1 ;
u(tache,temps)=1 & ng(tache)=40 > 0 & R(tache)=0 ;
nh(tache)=80 & nr(tache) = 100 ;
nh(tache)<nr(tache) ;
ListeGPU2[indiceListeGPU2].IndiceVM = 1 ;
indiceListeGPU2 = 2 ;
NombreTachesListeGPU2 = 2 ;

tache = 2 ;
u(tache,temps)=1 & ng(tache)=40 > 0 & R(tache)=0 ;
nh(tache)=80 & nr(tache) = 100 ;
nh(tache)>nr(tache) ;
ListeGPU2[indiceListeGPU2].IndiceVM = 2 ;
indiceListeGPU2 = 3 ;
NombreTachesListeGPU2 = 3 ;

tache = 3 ;
u(tache,temps)=1 & ng(tache)=0 0 & R(tache)=0 ;
nh(tache)=120 & nr(tache) = 100 ;
nh(tache)>nr(tache) ;
ListeCPU1[indiceListeCPU1].IndiceVM = 3 ;
indiceListeCPU1 = 1 ;
NombreTachesListeCPU1 = 1 ;

tache = 4 ;
u(tache,temps)=1 & R(tache)=1 ;
//on passe à la tâche suivante

tache = 5 ;
u(tache,temps)=1 & R(tache)=1 ;
//c'est la dernière tâche, on passe au t suivant

temps + 1 = 1 = ListeIntervalles[indice].BorneSup ;

tache = 0 ;
u(tache,temps)=1 & ng(tache)=40 > 0 & R(tache)=0 ;
nh(tache)=80 & nr(tache) = 100 ;
nh(tache)<nr(tache) ;
ListeGPU2[indiceListeGPU2].IndiceVM = 0 ;
indiceListeGPU2 = 1 ;
NombreTachesListeGPU2 = 1 ;

tache = 1 ;
u(tache,temps)=1 & ng(tache)=40 > 0 & R(tache)=0 ;

nh(tache)=80 & nr(tache) = 100 ;
nh(tache)<nr(tache) ;
ListeGPU2[indiceListeGPU2].IndiceVM = 1 ;
indiceListeGPU2 = 2 ;
NombreTachesListeGPU2 = 2 ;

tache = 2 ;
u(tache,temps)=1 & ng(tache)=40 > 0 & R(tache)=0 ;
nh(tache)=80 & nr(tache) = 100 ;
nh(tache)>nr(tache) ;
ListeGPU2[indiceListeGPU2].IndiceVM = 2 ;
indiceListeGPU2 = 3 ;
NombreTachesListeGPU2 = 3 ;

tache = 3 ;
u(tache,temps)=1 & ng(tache)=0 0 & R(tache)=0 ;
nh(tache)=120 & nr(tache) = 100 ;
nh(tache)>nr(tache) ;
ListeCPU1[indiceListeCPU1].IndiceVM = 3 ;
indiceListeCPU1 = 1 ;
NombreTachesListeCPU1 = 1 ;

tache = 4 ;
u(tache,temps)=1 & R(tache)=1 ;
//on passe à la tache suivante

tache = 5 ;
u(tache,temps)=1 & R(tache)=1 ;
//c'est la dernière la tâche et on est au dernier instant temps de l'intervalle étudié

Fin de l'algorithme.

On obtient alors les listes suivante pour le premier intervalle :

ListeGPU1			
ListeGPU2	0	1	2
ListeCPU1	3		
ListeCPU2			

En reprenant le même intervalle que celui pour l'exemple de l'algorithme précédent, le déroulement de celui ci est le suivant :

indice = 0 ;
temps = ListeIntervalles[indice].BorneInf = 0 ;

tache = 0 ;
u(tache,temps)=1 & R(tache)=0≠1 ;
//on passe la tâche suivant

tache = 1 ;
u(tache,temps)=1 & R(tache)=0≠1 ;
//on passe la tâche suivant

tache = 2 ;
u(tache,temps)=1 & R(tache)=0≠1 ;
//on passe la tâche suivant

Algorithme 6 Création de la liste des tâches préemptables

Précondition:

Entrée : indice de l'intervalle sur lequel on travaille.

Postcondition:

Sortie : liste des taches qui sont préemptables et qui doivent s'exécuter sur cet intervalle ainsi que le nombre de tâches préemptables qui doivent être exécutées sur l'intervalle.

```
1:  pour temps allant de ListeIntervalles[indice].BorneInf à ListeIntervalles[indice].BorneSup faire
2:      entier indiceListe = 0 ;
3:      entier NombreTachesPr = 0 ;
4:      pour tache allant de 0 à N() faire
5:          si u(tache,temps)=1 && R(tache)=1 alors
6:              ListeTachesPr[indiceListe].IndiceVM = tache ;
7:              indiceListe = indiceListe + 1 ;
8:              NombreTachesPr = NombreTachesPr + 1 ;
9:          fin si
10:     fin pour
11: fin pour
```

tache = 3 ;
u(tache,temps)=1 & R(tache)=0≠1 ;
//on passe la tâche suivant

tache = 4 ;
u(tache,temps)=1 & R(tache)=1 ;
ListeTachesPr[indiceListe].IndiceVM = 4 ; indiceListe = 1 ; NombreTachesPr = 1 ;

tache = 5 ;
u(tache,temps)=1 & R(tache)=1 ;
ListeTachesPr[indiceListe].IndiceVM = 5 ; indiceListe = 2 ; NombreTachesPr = 2 ; //toutes les tâches on été parcourue on passe à l'instant de temps suivant

temps + 1 = 1 = ListeIntervalles[indice].BorneSup ;

tache = 0 ;
u(tache,temps)=1 & R(tache)=0≠1 ;
//on passe la tâche suivant

tache = 1 ;
u(tache,temps)=1 & R(tache)=0≠1 ;
//on passe la tâche suivant

tache = 2 ;
u(tache,temps)=1 & R(tache)=0≠1 ;
//on passe la tâche suivant

tache = 3 ;
u(tache,temps)=1 & R(tache)=0≠1 ;
//on passe la tâche suivant

tache = 4 ;
u(tache,temps)=1 & R(tache)=1 ;
ListeTachesPr[indiceListe].IndiceVM = 4 ; indiceListe = 1 ; NombreTachesPr = 1 ;

tache = 5 ;
u(tache,temps)=1 & R(tache)=1 ;

ListeTachesPr[indiceListe].IndiceVM = 5 ; indiceListe = 2 ; NombreTachesPr = 2 ; //c'est la dernière la tâche et on est au dernier instant temps de l'intervalle étudié

Fin de l'algorithme.

Voici la liste résultante de cette exécution.

ListeTachesPr	4	5

Comme le déroulement de cet algorithme 19 est assez similaire que le premier de cette partie, à l'exception que les listes remplies sont celles des tâches préemptables. Je vais donc directement donner les listes issues de l'exécution.

ListePRGPU1	
ListePRGPU2	4
ListePRCPU1	5
ListePRCPU2	

Algorithme 7 Création des listes des tâches préemptables

Précondition:

Entrée : indice de l'intervalle sur lequel on travaille.

Postcondition:

Sortie : les quatres listes des tâches préemptables répartie en fonction de leurs besoins

```
1: pour temps allant de ListeIntervalles[indice].BorneInf à ListeIntervalles[indice].BorneSup faire
2:     entier indiceListePRGPU1 = 0 ;
3:     entier indiceListePRGPU2 = 0 ;
4:     entier indiceListePRCPU1 = 0 ;
5:     entier indiceListePRCPU2 = 0 ;
6:     entier NombreTachesListePRGPU1 = 0 ;
7:     entier NombreTachesListePRGPU2 = 0 ;
8:     entier NombreTachesListePRCPU1 = 0 ;
9:     entier NombreTachesListePRCPU2 = 0 ;
10:    pour tache allant de 0 à NombreTachesPr faire
11:        si u(ListeTachesPr[tache].IndiceVM,temps)=1 && R(ListeTachesPr[tache].IndiceVM)=0 alors
12:            si nh(ListeTachesPr[tache].IndiceVM)>nr(ListeTachesPr[tache].IndiceVM) alors
13:                ListePRGPU1[indiceListePRGPU1].IndiceVM = ListeTachesPr[tache].IndiceVM ;
14:                indiceListePRGPU1 = indiceListePRGPU1 + 1 ;
15:                NombreTachesListePRGPU1 = NombreTachesListePRGPU1 + 1 ;
16:            sinon
17:                ListePRGPU2[indiceListePRGPU2].IndiceVM = ListeTachesPr[tache].IndiceVM ;
18:                indiceListePRGPU2 = indiceListePRGPU2 + 1 ;
19:                NombreTachesListePRGPU2 = NombreTachesListePRGPU2 + 1 ;
20:            fin si
21:        sinon
22:            si u(ListeTachesPr[tache].IndiceVM,temps)=1 && R(ListeTachesPr[tache].IndiceVM)=1 alors
23:                si nh(ListeTachesPr[tache].IndiceVM)>nr(ListeTachesPr[tache].IndiceVM) alors
24:                    ListePRCPU1[indiceListePRCPU1].IndiceVM = ListeTachesPr[tache].IndiceVM ;
25:                    indiceListePRCPU1 = indiceListePRCPU1 + 1 ;
26:                    NombreTachesListePRCPU1 = NombreTachesListePRCPU1 + 1 ;
27:                sinon
28:                    ListePRCPU2[indiceListePRCPU2].IndiceVM = ListeTachesPr[tache].IndiceVM ;
29:                    indiceListePRCPU2 = indiceListePRCPU2 + 1 ;
30:                    NombreTachesListePRCPU2 = NombreTachesListePRCPU2 + 1 ;
31:                fin si
32:            fin si
33:        fin si
34:    fin pour
35: fin pour
```

3.2.4 Algorithmes de calcul priorité des taches

Maintenant que les algorithmes qui permettent de construire les différentes listes de tâches à affecter, ont été décrits précédemment. Il reste à ordonner ces listes afin d'affecter les machines virtuelles de manière à respecter certaines contraintes.

Ces contraintes à respecter sont :
- Une machine virtuelle ne peut être affectée que sur une machine physique tel que la matrice de préaffectation l'autorise.
- Une machine virtuelle ne peut être migrée d'une machine physique à une autre, uniquement si cette dernière a été exécutée sur la machine physique assez longtemps pour que la durée d'exécution soit supérieure ou égale à la durée de migration mt_i
- Pour les machines virtuelles préemptable, si la machine virtuelle en question à été suspendue dans un intervalle de temps précédent elle ne peut être remis en route que si le temps de suspension est supérieur ou égale au temps de "resume" rti, j

Afin de gérer tous ces cas de figure, chaque machine virtuelle se vera attribué une priorité. Pour calculer cette priorité on définit les variables suivantes :
- $IB_{i,k}$ qui prend pour valeur le nombre de machines physiques M si la tâche i était affectée sur la machine k sur l'intervalle précédent et 0 sinon
- WG_i qui prend pour valeur le nombre de machines qui ne peuvent pas recevoir la tâche i sur le premier intervalle. Sur les autres intervalles WG_i prendra une valeur négative si la tâche ne peut pas être migrée et sinon prendra le nombre de machine qui ne peut pas recevoir la tâche i

Voici l'algorithme permettant de calculer les priorités des tâches de la liste des tâches non préemptable ayant des besoins en terme de GPU et CPU et tel que leurs besoins en HDD > besoins en RAM. L'algorithme pour le calcul des priorités pour les autres listes des tâches non préemptables ainsi que les listes des tâches préemptables fonctionne de la même manière.

Pour donner un exemple de déroulement, je vais rester sur l'intervalle 0 sur lequel on a construit les listes dans la partie précédente. Je vais dérouler l'algorithme sur la ListeGPU2 comme c'est celle qui contient le plus de tâche avec comme indiceServeur égale à 2 et je donnerai le tableau résumant tout les priorités.

indice = 0 ;
IB = 0 ;
WG = 0 ;
$IndiceDeBoucle = 0$
MachineRecevoir = 0 ;
//Comme c'est le premier intervalle, la tâche ListeGPU1[IndiceDeBoucle].IndiceVM = 0 n'était pas affectée sur un serveur.
IB = 0 ;
$IndiceDeBoucle2 = 0$
q(0,0)=1 alors MachineRecevoir = 1 ;
$IndiceDeBoucle2 = 1$
q(0,1)=0 alors MachineRecevoir = 1 ;
$IndiceDeBoucle2 = 2$
q(0,2)=0 alors MachineRecevoir = 1 ;
WG = 3 - 1 = 2 ;
ListeGPU1[IndiceDeBoucle].Prio = 0 + 2 = 2 ;

$IndiceDeBoucle = 1$
MachineRecevoir = 0 ;

Algorithme 8 Calcul Priorité des taches de la ListeGPU1

Précondition:

Entrée : indice de l'intervalle et indiceServeur qui correspond à l'indice du serveur sur lequel on cherche à affecter des machines virtuelles.

Postcondition:

Sortie : la liste des tâches à affecter classée par ordre décroissant des priorités sur l'intervalle pour la machine physique donnée.

```
1: entier IB = 0 ;
2: entier WG = 0 ;
3: entier MachineRecevoir ;
4: pour IndiceDeBoucle allant de 0 à NombreTachesListeGPU1 faire
5:         MachineRecevoir = 0 ;
6:         si la tâche ListeGPU1[IndiceDeBoucle].IndiceVM était affectée sur indiceServeur à l'intervalle précédent alors
7:                 IB = M() ;
8:         sinon
9:                 IB = 0 ;
10:        fin si
11:        si indice ≠ 0 alors
12:                si La durée d'execution de la tâche < Mt_i alors
13:                        si la tâche n'était pas exécutée à l'intervalle précédent alors
14:                                on chercher le dernier intervalle sur lequel la tâche a été exécutée
15:                                si sur l'intervalle trouvé la tâche a été exécutée sur IndiceServeur alors
16:                                        WG = IndiceServeur ;
17:                                sinon
18:                                        WG = -M()
19:                                fin si
20:                        sinon
21:                                si sur l'intervalle précédent la tâche été exécutée sur IndiceServeur alors
22:                                        WG = IndiceServeur
23:                                sinon
24:                                        WG = -M() ;
25:                                fin si
26:                        fin si
27:                sinon
28:                        pour IndiceDeBoucle2 allant de 0 à M() - 1 faire
29:                                si si la machine IndiceDeBoucle2 peut recevoir la tâche alors
30:                                        MachineRecevoir++ ;
31:                                fin si
32:                        fin pour
33:                        WG = M() - MachineRecevoir ;
34:                fin si
35:        sinon
36:                pour IndiceDeBoucle2 allant de 0 à M() - 1 faire
37:                        si si la machine IndiceDeBoucle2 peut recevoir la tâche alors
38:                                MachineRecevoir++ ;
39:                        fin si
40:                fin pour
41:                WG = M() - MachineRecevoir ;
42:        fin si
43:        ListeGPU1[IndiceDeBoucle].Prio = IB + WG ;
44: fin pour
45: Trier(ListeGPU1,Ordre Décroissant Prio) ;
```

//Comme c'est le premier intervalle, la tâche ListeGPU1[IndiceDeBoucle].IndiceVM = 1 n'était pas affectée sur un serveur.
IB = 0 ;
$IndiceDeBoucle2 = 0$
q(1,0)=1 alors MachineRecevoir = 1 ;
$IndiceDeBoucle2 = 1$
q(1,1)=1 alors MachineRecevoir = 2 ;
$IndiceDeBoucle2 = 2$
q(1,2)=1 alors MachineRecevoir = 3 ;
WG = 3 - 3 = 0 ;
ListeGPU1[IndiceDeBoucle].Prio = 0 + 0 = 0 ;

$IndiceDeBoucle = 2$
MachineRecevoir = 0 ;
//Comme c'est le premier intervalle, la tâche ListeGPU1[IndiceDeBoucle].IndiceVM = 2 n'était pas affectée sur un serveur.
IB = 0 ;
$IndiceDeBoucle2 = 0$
q(2,0)=1 alors MachineRecevoir = 1 ;
$IndiceDeBoucle2 = 1$
q(2,1)=1 alors MachineRecevoir = 2 ;
$IndiceDeBoucle2 = 2$
q(2,2)=0 alors MachineRecevoir = 2 ;
WG = 3 - 2 = 0 ;
ListeGPU1[IndiceDeBoucle].Prio = 0 + 1 = 1 ;

//on parcourue tout la liste des tâches. /Fin de l'algo.

Une fois toutes les priorités calculées pour toutes les listes et ces dernières sont triés, ont obtient alors les résultats suivants :

Tâches ListeGPU1			
Prio Tâches			
Tâches ListeGPU2	0	2	1
Prio Tâches	2	1	0
Tâches ListeCPU1	3		
Prio Tâches	2		
Tâches ListeCPU2			
Prio Tâches			
Tâches ListePRGPU1			
Prio Tâches			
Tâches ListePRGPU2	4		
Prio Tâches	2		
Tâches ListePRCPU1	5		
Prio Tâches	2		
Tâches ListePRCPU2			
Prio Tâches			

3.2.5 Affectation des tâches sur les machines physiques

Les différents algorithmes précédemment présentés permettent de construire les différentes listes qui vont permettre de d'effectuer la répartition des machines virtuelles sur les machines physiques. Dans cette partie, je vais présenter les algorithmes qui permettent d'effectuer cette répartition. Pour cela, nous allons distinguer le cas des tâches non préemptable de celles qui le sont. Cependant l'ensemble des algorithmes permettant de construire l'ordonnancement complet sont appelés dans l'algorithme suivant.

Algorithme 9 Ordonnancement

Précondition:
Entrée : indice de l'intervalle et indiceServeur qui correspond à l'indice du serveur sur lequel on cherche à affecter des machines virtuelles.

Postcondition:
Sortie : le tableau ListOrdo représentant la matrice d'affectation des machines virtuelles sur les machines physique

```
1: entier CPU,GPU,RAM,HDD ;
2: entier iboucleS ;
3: entier indiceTab ;
4: entier temps ;
5: pour iboucleS allant de 0 à M()-1 faire
6:         entier indiceS = ListeServeur[iboucleS].IndiceServeur ;
7:         CPU = ListeServeur[indiceS].CPU ;
8:         GPU = ListeServeur[indiceS].GPU ;
9:         RAM = ListeServeur[indiceS].RAM ;
10:        HDD = ListeServeur[indiceS].HDD ;
11:        si (NombreTachesListeGPU1 ≠ 0) && (NombreTachesListeGPU2 ≠ 0) alors
12:                Calcul Priorité des taches de la ListeGPU1(indice,indiceS) ;
13:                Calcul Priorité des taches de la ListeGPU2(indice,indiceS) ;
14:                Algorithme d'affectation tâche GPU(indice) ;
15:        fin si
16:        si (NombreTachesListeCPU1 ≠ 0) && (NombreTachesListeCPU2 ≠ 0) alors
17:                Calcul Priorité des taches de la ListeCPU1(indice,indiceS) ;
18:                Calcul Priorité des taches de la ListeCPU2(indice,indiceS) ;
19:                Algorithme d'affectation tâche CPU(indice) ;
20:        fin si
21:        si (NombreTachesListePRGPU1 ≠ 0) && (NombreTachesListePRGPU2 ≠ 0) alors
22:                Algorithme d'affectation tâche préemptable sur serveur allumé(indice,indiceS) ;
23:        fin si
24:        si (NombreTachesListePRCPU1 ≠ 0) && (NombreTachesListePRCPU2 ≠ 0) alors
25:                Algorithme d'affectation tâche préemptable sur serveur allumé(indice,indiceS) ;
26:        fin si
27:        ListeServeur[indiceS].CPU = CPU ;
28:        ListeServeur[indiceS].GPU = GPU ;
29:        ListeServeur[indiceS].RAM = RAM ;
30:        ListeServeur[indiceS].HDD = HDD ;
31: fin pour
```

Affectation des tâches non préemptables

Pour l'affectation des tâches non préemptables, il y a un certain ordre logique à respecter. Effectivement, toutes les machines physiques ne possède pas forcement des GPUs mais posséderons forcement des CPUs, de façon générale un serveur possède souvent plus de capacité en RAM que en disque dur. C'est pourquoi on va d'abord affecter les tâches ayant besoins de CPU et GPU, c'est à dire les tâches de ListeGPU1 et ListeGPU2 pour être exécutées, comme la capacité de disque dur d'un serveur est plus faible que la capacité en RAM, on commencera à affecter d'abord les tâches ayant des besoins HDD > besoins RAM avant d'affecter celle qui ont des besoins RAM > besoins HDD.

La première idée d'algorithme pour réaliser l'affection est la suivante :

Algorithme 10 Algorithme d'affectation tâche GPU

Précondition:

Entrée : indice de l'intervalle et indiceServeur qui correspond à l'indice du serveur sur lequel on cherche à affecter des machines virtuelles.

Postcondition:

Sortie : le tableau deux dimension représentant la matrice d'ordonnancement des machines virtuelles sur les machines physiques.

```
1:  entier indiceBoucle ;
2:  entier instantT = ListeIntervalle[indice].BorneInf ;
3:  pour indiceBoucle allant de 0 à NombreTachesListeGPU1 faire
4:          entier indiceVM = ListeGPU1[indiceBoucle].IndiceVM ;
5:          si ListOrdo[instantT][indiceVM].affecter≠1 alors
6:                  si mc(indiceServeur)>nc(indiceVM) alors
7:                          si mh(indiceServeur)>nh(indiceVM) alors
8:                                  si mr(indiceServeur)>nr(indiceVM) alors
9:                                          ListOrdo[instantT][indiceVM].affecter = 1 ;
10:                                         ListOrdo[instantT][indiceVM].IndiceMachine = IndiceServeur ;
11:                                         MaJServeur(indiceVM,indice,indiceServeur) ;
12:                                 fin si
13:                         fin si
14:                 fin si
15:         fin si
16: fin pour
17: pour indiceBoucle allant de 0 à NombreTachesListeGPU2 faire
18:         entier indiceVM = ListeGPU2[indiceBoucle].IndiceVM ;
19:         si ListOrdo[instantT][indiceVM].affecter≠1 alors
20:                 si mc(indiceServeur)>nc(indiceVM) alors
21:                         si mh(indiceServeur)>nh(indiceVM) alors
22:                                 si mr(indiceServeur)>nr(indiceVM) alors
23:                                         ListOrdo[instantT][indiceVM].affecter = 1 ;
24:                                         ListOrdo[instantT][indiceVM].IndiceMachine = IndiceServeur ;
25:                                         MaJServeur(indiceVM,indice,indiceServeur) ;
26:                                 fin si
27:                         fin si
28:                 fin si
29:         fin si
30: fin pour
```

L'algorithme pour les tâches non préemptables ayant uniquement besoins de CPU suit le même raisonnement.

Algorithme 11 Algorithme d'affectation tâche CPU

Précondition:

Entrée : indice de l'intervalle et indiceServeur qui correspond à l'indice du serveur sur lequel on cherche à affecter des machines virtuelles.

Postcondition:

Sortie : le tableau deux dimension représentant la matrice d'ordonnancement des machines virtuelles sur les machines physiques.

```
1: entier indiceBoucle ;
2: entier instantT = ListeIntervalle[indice].BorneInf ;
3: pour indiceBoucle allant de 0 à NombreTachesListeCPU1 faire
4:         entier indiceVM = ListeCPU1[indiceBoucle].IndiceVM ;
5:         si ListOrdo[instantT][indiceVM].affecter≠1 alors
6:                 si mc(indiceServeur)>nc(indiceVM) alors
7:                         si mh(indiceServeur)>nh(indiceVM) alors
8:                                 si mr(indiceServeur)>nr(indiceVM) alors
9:                                         ListOrdo[instantT][indiceVM].affecter = 1 ;
10:                                        ListOrdo[instantT][indiceVM].IndiceMachine = IndiceServeur ;
11:                                        MaJServeur(indiceVM,indice,indiceServeur) ;
12:                                fin si
13:                        fin si
14:                fin si
15:        fin si
16: fin pour
17: pour indiceBoucle allant de 0 à NombreTachesListeCPU2 faire
18:        entier indiceVM = ListeCPU2[indiceBoucle].IndiceVM ;
19:        si ListOrdo[instantT][indiceVM].affecter≠1 alors
20:                si mc(indiceServeur)>nc(indiceVM) alors
21:                        si mh(indiceServeur)>nh(indiceVM) alors
22:                                si mr(indiceServeur)>nr(indiceVM) alors
23:                                        ListOrdo[instantT][indiceVM].affecter = 1 ;
24:                                        ListOrdo[instantT][indiceVM].IndiceMachine = IndiceServeur ;
25:                                        MaJServeur(indiceVM,indice,indiceServeur) ;
26:                                fin si
27:                        fin si
28:                fin si
29:        fin si
30: fin pour
```

Algorithme 12 Algorithme MaJServeur

Précondition:

Entrée : instant de temps et indiceServeur qui correspond à l'indice du serveur sur lequel on cherche à affecter des machines virtuelles et indiceVM l'indice de la VM affecter à ce serveur. Il y a aussi le tableau à deux dimensions ListeServeur[HorizonPlanification][NombreServeur] contentant les caractéristiques des serveurs.

Postcondition:

Sortie : Le tableau ListeServeur mis à jour après affectation de la VM sur le serveur indiceServeur.

```
1: ListeServeur[temps][indiceServeur].GPU = ListeServeur[temps][indiceServeur].GPU -
   ng(indiceVM) ;
2: ListeServeur[temps][indiceServeur].CPU = ListeServeur[temps][indiceServeur].CPU -
   nc(indiceVM) ;
3: ListeServeur[temps][indiceServeur].RAM = ListeServeur[temps][indiceServeur].RAM -
   nr(indiceVM) ;
4: ListeServeur[temps][indiceServeur].HDD = ListeServeur[temps][indiceServeur].HDD -
   nh(indiceVM) ;
```

Cependant, au cours du projet cet algorithme à évolué afin de prendre en compte le réseau ainsi que le temps d'exécution des tâches afin de gérer la migration de machine virtuelle. L'algorithme devient alors le suivant :

Algorithme 13 Algorithme d'affectation tâche GPU avec gestion réseau et migration

Précondition:

Entrée : indice de l'intervalle et indiceServeur qui correspond à l'indice du serveur sur lequel on cherche à affecter des machines virtuelles.

Postcondition:

Sortie : le tableau deux dimension représentant la matrice d'ordonnancement des machines virtuelles sur les machines physiques.

```
1: entier indiceBoucle ;
2: entier instantT = ListeIntervalle[indice].BorneInf ;
3: entier instantT2 = ListeIntervalle[indice].BorneSup ;
4: pour indiceBoucle allant de 0 à NombreTachesListeGPU1 faire
5:         entier indiceVM = ListeGPU1[indiceBoucle].IndiceVM ;
6:         si ListOrdo[instantT][indiceVM].affecter≠1 alors
7:                 si mg(indiceServeur)>ng(indiceVM) alors
8:                         si mc(indiceServeur)>nc(indiceVM) alors
9:                                 si mh(indiceServeur)>nh(indiceVM) alors
10:                                        si mr(indiceServeur)>nr(indiceVM) alors
11:                                                pour indiceBoucle2 allant de 0 à N() faire
12:                                                        si a(indiceVM,indiceBoucle)=1 alors
13:                                                                GestionReseau(indiceVM,indiceBoucle,indice,indiceSer
14:                                                        fin si
15:                                                fin pour
16:                                                pour T allant de instantT à instantT2 faire
17:                                                        si ListOrdo[instantT][indiceVM].affecter≠1
                                                           alors
18:                                                                ListOrdo[T][indiceVM].affecter = 1 ;
19:                                                                ListOrdo[T][indiceVM].IndiceMachine =
                                                                   IndiceServeur ;
20:                                                                ListeGPU1[indiceVM].dureeExe = Lis-
                                                                   teGPU1[indiceVM].dureeExe + 1
21:                                                                MaJServeur(indiceVM,T,indiceServeur) ;
22:                                                        fin si
23:                                                fin pour
24:                                        fin si
25:                                fin si
26:                        fin si
27:                fin si
28:        fin si
29: fin pour
```

Gestion réseau et migration

La fonction GestionReseau(indiceVM1,indiceVM2,indice,indiceServeur) a pour but de vérifier si le réseau permet d'affecter deux machines qui ont une affinité entre elle sur deux serveurs différents. Pour cela elle fait appel au deux fonctions qui sont quasi similaires à l'exception qu'une sert à tester si le réseau permet l'affection et que l'autre met à jour le réseau.

Algorithme 14 GestionReseau

Précondition:

Entrée : indice de l'intervalle et indiceServeur qui correspond à l'indice du serveur sur lequel on cherche à affecter des machines virtuelles, ainsi que les indices indiceVM1 et indiceVM2 deux machines virtuelles qui rentrent en communication. indiceVM1 indique la machine que l'on est en train d'affecter et indiceVM2 la machine avec laquelle elle possède une affinité.

Postcondition:

Sortie : affecte la machine virtuelle comme non affectée si le réseau ne permet pas l'affectation.

```
1: entier instantT = ListeIntervalle[indice].BorneInf ;
2: entier instantT2 = ListeIntervalle[indice].BorneSup ;
3: si ListOrdo[instanT][indiceVM2].affecter=1 alors
4:         si CalculFesabiliteReseau(indiceVM1,indiceVM2,indice,indiceServeur,indiceServeurVM2)=
           0 alors
5:                 pour T allant de instantT à instantT2 faire
6:                         ListOrdo[T][indiceVM].affecter = 1 ;
7:                         ListOrdo[T][indiceVM].IndiceMachine = -1 ;
8:                 fin pour
9:         sinon
10:                MaJReseau(indiceVM1,indiceVM2,indice,indiceServeurVM2) ;
11:        fin si
12: fin si
```

Algorithme 15 CalculFesabiliteReseau

Précondition:

Entrée : indice de l'intervalle ,les indices indiceVM1 et indiceVM2 tel que indiceVM1 indique la machine que l'on est en train d'affecter et indiceVM2 la machine avec laquelle elle possède une affinité, ainsi que indiceServeur à l'indice du serveur sur lequel on cherche à affecter des machines virtuelles et indiceServeurVM2 l'indice du serveur où est affecter VM2.

Postcondition:

Sortie : retourne 1 si le réseau permet la communication et 0 sinon.

```
1: entier iArc, iTime ;
2: entier instantT = ListeIntervalle[indice].BorneInf ;
3: entier instantT2 = ListeIntervalle[indice].BorneSup ;
4: entier memoireBdW ;
5: pour iArc allant de 0 à NbArc()-1  faire
6:         pour iTime allant de instantT à instantT2  faire
7:                 entier iBdW = 0 ;
8:                 entier iBoucle ;
9:                 pour iBoucle allant de 0 à NbMachArc(iArc) faire
10:                        CoupleMachines(iArc,iBoucle,indiceServeur,indiceServeurVM2) ;
11:                        iBdW = b(indiceVM1,indiceVM2) ;
12:                        ListReseau[iTime][iEdge][iBoucle].Mach1 = indiceServeur ;
13:                        ListReseau[iTime][iEdge][iBoucle].Mach1 = indiceServeurVM2 ;
14:                        memoireBdW = ListReseau[iTime][iEdge][iBoucle].BdePassandeDispo ;
15:                        ListReseau[iTime][iEdge][iBoucle].BdePassandeDispo = ListReseau[iTime][iEdge][iBoucle].BdePassandeDispo - iBdw ;
16:                        si ListReseau[iTime][iEdge][iBoucle].BdePassandeDispo > 0 alors
17:
18:                                return  1 ;
19:                        sinon
20:
21:                                return  0 ;
22:                        fin si
23:                        ListReseau[iTime][iEdge][iBoucle].BdePassandeDispo = memoireBdW ;
24:                fin pour
25:        fin pour
26: fin pour
```

Algorithme 16 MaJReseau

Précondition:

Entrée : indice de l'intervalle ,les indices indiceVM1 et indiceVM2 tel que indiceVM1 indique la machine que l'on est en train d'affecter et indiceVM2 la machine avec laquelle elle possède une affinité, ainsi que indiceServeur à l'indice du serveur sur lequel on cherche à affecter des machines virtuelles et indiceServeurVM2 l'indice du serveur où est affecter VM2.

Postcondition:

Sortie : retourne 1 si le réseau permet la communication et 0 sinon.

```
1: entier iArc, iTime ;
2: entier instantT = ListeIntervalle[indice].BorneInf ;
3: entier instantT2 = ListeIntervalle[indice].BorneSup ;
4: pour iArc allant de 0 à NbArc()-1 faire
5:         pour iTime allant de instantT à instantT2 faire
6:                 entier iBdW = 0 ;
7:                 entier iBoucle ;
8:                 pour iBoucle allant de 0 à NbMachArc(iArc) faire
9:                         CoupleMachines(iArc,iBoucle,indiceServeur,indiceServeurVM2) ;
10:                        iBdW = b(indiceVM1,indiceVM2) ;
11:                        ListReseau[iTime][iEdge][iBoucle].Mach1 = indiceServeur ;
12:                        ListReseau[iTime][iEdge][iBoucle].Mach1 = indiceServeurVM2 ;
13:                        ListReseau[iTime][iEdge][iBoucle].BdePassandeDispo = ListReseau[iTime][iEdge][iBoucle].BdePassandeDispo - iBdw ;
14:                fin pour
15:        fin pour
16: fin pour
```

Maintenant que tout les algorithmes permettant de réaliser l'affectation des tâches non préemptables ont été présentés, je vais illustrer son déroulement avec les données utilisées et calculées précédemment. Voici un rappel des priorités des tâches calculées pour le premier serveur de la liste des serveurs.

Tâches ListeGPU1			
Prio Tâches			
Tâches ListeGPU2	0	2	1
Prio Tâches	2	1	0
Tâches ListeCPU1	3		
Prio Tâches	2		
Tâches ListeCPU2			
Prio Tâches			
Tâches ListePRGPU1			
Prio Tâches			
Tâches ListePRGPU2	4		
Prio Tâches	2		
Tâches ListePRCPU1	5		
Prio Tâches	2		
Tâches ListePRCPU2			
Prio Tâches			

On commence par affecter les tâches issues des listes GPU. Ici, seule la ListeGPU2 contient des tâches à affecter. On utilise l'algorithme Algorithme d'affectation tâche GPU avec gestion réseau et migration 13. Je vais décrire le déroulement qui se produit.

instantT = 0 ;
instantT2 = 0 ;
indiceBoucle = 0 ;
indiceVM = 0 ;
mg(2) = 150 & ng(0) = 10 donc mg(2)>ng(0) ;
mc(2) = 250 & nc(0) = 40 donc mc(2)>nc(0) ;
mr(2) = 1000 & nr(0) = 100 donc mr(2)>nr(0) ;
mh(2) = 250 & nr(0) = 80 donc mh(2)>nh(0) ;
//pas d'affinité entre la tâche 0 et les autres
T = 0 ; ListOrdo[0][0].affecter = 1 ;
ListOrdo[0][0].IndiceMachine = 2 ;
ListeGPU2[0].dureeExe = 0 + 1 = 1 ; MaJServer(0,0,2) ;
Nouvelle valeur du serveur 2 : GPU = 140 CPU = 210 RAM = 900 HDD = 170 ;

T = 1 ; ListOrdo[0][0].affecter = 1 ;
ListOrdo[0][0].IndiceMachine = 2 ;
ListeGPU2[0].dureeExe = 1 + 1 = 2 ; MaJServer(0,1,2) ;
Nouvelle valeur du serveur 2 : GPU = 140 CPU = 210 RAM = 900 HDD = 170 ;

indiceBoucle = 1 ;
indiceVM = 1 ;
mg(2) = 140 & ng(1) = 10 donc mg(2)>ng(1) ;
mc(2) = 210 & nc(1) = 40 donc mc(2)>nc(1) ;

mr(2) = 900 & nr(1) = 100 donc mr(2)>nr(1) ;
mh(2) = 170 & nr(1) = 80 donc mh(2)>nh(1) ;
//pas d'affinité entre la tâche 1 et les autres
T = 0 ; ListOrdo[0][1].affecter = 1 ;
ListOrdo[0][1].IndiceMachine = 2 ;
ListeGPU2[1].dureeExe = 0 + 1 = 1 ; MaJServer(1,0,2) ;
Nouvelle valeur du serveur 2 : GPU = 130 CPU = 170 RAM = 800 HDD = 90 ;

T = 1 ; ListOrdo[0][0].affecter = 1 ;
ListOrdo[0][0].IndiceMachine = 2 ;
ListeGPU2[0].dureeExe = 1 + 1 = 2 ; MaJServer(1,1,2) ;
Nouvelle valeur du serveur 2 : GPU = 130 CPU = 170 RAM = 800 HDD = 90 ;

indiceBoucle = 2 ;
indiceVM = 2 ;
mg(2) = 130 & ng(2) = 10 donc mg(2)>ng(1) ;
mc(2) = 170 & nc(2) = 40 donc mc(2)>nc(1) ;
mr(2) = 800 & nr(2) = 100 donc mr(2)>nr(1) ;
mh(2) = 90 & nr(2) = 80 donc mh(2)>nh(1) ;
//affinité entre la tâche 2 et la tâche 3
//la tâche 3 n'est pas affectée donc pas de problème réseau
T = 0 ; ListOrdo[0][2].affecter = 1 ;
ListOrdo[0][2].IndiceMachine = 2 ;
ListeGPU2[2].dureeExe = 0 + 1 = 1 ; MaJServer(2,0,2) ;
Nouvelle valeur du serveur 2 : GPU = 120 CPU = 130 RAM = 700 HDD = 10 ;

T = 1 ; ListOrdo[0][0].affecter = 1 ;
ListOrdo[0][0].IndiceMachine = 2 ;
ListeGPU2[0].dureeExe = 1 + 1 = 2 ; MaJServer(1,1,2) ;
Nouvelle valeur du serveur 2 : GPU = 120 CPU = 130 RAM = 700 HDD = 10 ;
//plus de tâche avec des besoins en GPU à affecter. //Fin de l'exécution de l'algorithme.

Toutes les tâches ayant des besoins en GPU sont affectées on passe maintenant au tâche avec uniquement des besoins en CPU. Ici on a une seule tâche dans ce cas mais le serveur 2 ne peut pas la recevoir car plus assez de disque dur disponible. Donc elle ira sur la machine 0, comme c'est un tâche qui communique avec la tâche 2 mais que c'est le deux seule tâches qui communiquent entre elles, il n'y a aucun problème.

On a donc alors pour les instants $t = 0$ et$t = 1$, l'ordonnancement suivante pour les tâches non préemptable.

$$\begin{matrix} 2 & 2 & 2 & 0 \\ 2 & 2 & 2 & 0 \end{matrix}$$

Affectation des tâches préemptables

Pour l'affectation des tâches préemptables, le raisonnement est un peu différent. En effet pusiqu'elles sont préemptables on peut ne pas les exécuter si aucun serveur possède plus assez de ressources après l'affectation des tâches non préemptables.

Le raisonnement est alors le suivant : on essaye d'affecter un maximum de tâches préemtables sur les machines déjà allumées toujours en suivant le même principe concernant la priorité des listes. Cependant, on affecte la tâches uniquement si le min des couts d'affectations de la tâche i sur les machines j pouvant accueillir la tâche i est inférieur à la pénalité de ne pas exécuter cette dernière. Si il reste des tâches non affectées, on regarde s'il est intéresant de rallumer une machine physique parmis l'ensemble des machines restantes qui sont éteintes, noté $\bar{M}$.
Pour savoir si on rallume une machine on cherche l'indiceServeur tel que indiceServeur $= \arg\min_{j \in \bar{M}}(gaj)$, où $gaj = \sum_{t=t1}^{t2} \beta t + (t2 - t1) \times \sum_{i \in \bar{M}}(ct_{i,j} - \rho_i)$. Si on trouve qu'il faut pas rallumé de serveur alors l'affectation est finie.

Voici les algorithmes qui permettent de réaliser ces opérations. Le premier correspond à la fonction qui permet d'appeler la fonction qui affecte les VMs sur les serveurs allumé et la fonction qui permet de rallumer un serveur.

Algorithme 17 Algorithme d'affectation tâche préemptable sur serveur allumé

Précondition:
Entrée : indice de l'intervalle sur lequel on travaille et le nombre de serveur allume noté NombreDeServeurAllume

Postcondition:
Sortie : le tableau deux dimension représentant la matrice d'ordonancement des machines virtuelles sur les machines physiques.

```
1:  pour indiceBoucle allant de 0 à NombreDeServeurAllume faire
2:          OrdoGPUPr(indice,ListeServeurON[indiceBoucle].indiceServeur) ;
3:  fin pour
4:  si NombreTrachesPr > 0 && NombreDeServeurAllume ≤ M() alors
5:          AllumageMachine(indice) ;
6:  fin si
7:  pour indiceBoucle allant de 0 à NombreDeServeurAllume faire
8:          OrdoCPUPr(indice,ListeServeurON[indiceBoucle].indiceServeur) ;
9:  fin pour
10: si NombreTrachesPr > 0 && NombreDeServeurAllume ≤ M() alors
11:         AllumageMachine(indice) ;
12: fin si
```

La fonction OrdoGPUPr et OrdoCPUPr sont des fonctions qui permettent de déterminer quelle machine physique à un cout min pour les tâches à placer, une fois l'indiceServeur déterminé elle permet de créer la liste des tâches à affecter sur la machine indiceServeur. On appelle alors la fonction qui permet de calculer les priorités des tâches de la liste et après on affecte sur la machine physique à l'aide de la même fonction que pour les tâches non préemptables 10 & 11.

On va maintenant s'intéresser un peu plus particulièrement à l'algorithme qui permet de déterminer si il faut rallumer une machine physique.

Dans l'algorithme précédent, on définit l'indice de la machine qui va être rallumée, il faut refaire des listes des tâches qui peuvent être affectées sur cette machine avant de passer à l'affectation des tâches sur la nouvelle machine allumée. On définit alors quatre listes qui sont ListePRGPU1Machinej, ListePRGPU2Machinej, ListePRCPU1Machinej et ListePRCPU2Machinej

Algorithme 18 Algorithme pour déterminer s'il faut allumer une nouvelle machine

Précondition:

Entrée : indice de l'intervalle sur lequel on travaille et le nombre de serveur allume noté NombreDeServeurAllume

Postcondition:

Sortie : indiceServeur de la machine qu'il faut rallumer.

```
1: entier iboucle, iboucle1 ;
2: entier indiceTab = 0 ;
3: entier temps,MachineRallume ;
4: entier t1, t2 ;
5: réel gain ;
6: pour iboucle allant de NombreDeServeurAllume à M()-1 faire
7:         ListeDegaj[indiceTab].indiceServeur = ListeServeur[iboucle].indiceServeur ;
8:         gain = 0 ;
9:         pour iboucle1 allant de 0 à NombreTachesPr - 1 faire
10:                indiceVM = ListeTachesPr[iboucle1].indiceVM ;
11:                ServeurPossible = ListeServeur[iboucle].indiceServeur ;
12:                si q(indiceVM,ServeurPossible)=1 alors
13:                        t1 = ListeIntervalles[indice].BorneInf ;
14:                        t2 = ListeIntervalles[indice].BorneSup ;
15:                        pour temps allant de t1 à t2 faire
16:                                gain = gain + β(ServeurPossible)
17:                        fin pour
18:                        si (CalculCoutAffectation(indiceVM,ServeurPossible)<ρ(ServeurPossible))
                           alors
19:                                gain = gain + (t2-t1)*CalculCoutAffectation(indiceVM,ServeurPossible) ;
20:                        fin si
21:                sinon
22:                        gain = 0 ;
23:                fin si
24:        fin pour
25:        ListeDegaj[indiceTab].gain = gain ;
26:        indiceTab++ ;
27: fin pour
28: Trie par ordre croissant des gains du tableau ListeDegaj ;
29: si ListeDegaj[0].gain < 0 alors
30:        MachineRallume = ListeDegaj[0].indiceServeur ;
31:        NombreDeServeurAllume++ ;
32:        ConstructionListesTachePrMachineON(indice,MachineRallume) ;
33:        si (NbTacheGPU1_j > 0)||(NbTacheGPU2_j > 0) alors
34:                AffectationgGPUPre(indice,MachineRallume) ;
35:        fin si
36:        si (NbTacheCPU1_j > 0)||(NbTacheCPU2_j > 0) alors
37:                AffectationCPUPre(indice,MachineRallume) ;
38:        fin si
39: fin si
```

Algorithme 19 Création des listes des tâches préemptables

Précondition:

Entrée : indice de l'intervalle sur lequel on travaille et indiceServeur qui correspond à l'indice du serveur que l'on a choisi de rallumé.

Postcondition:

Sortie : les quatre listes des tâches préemptables qui peuvent être affectées à la machine rallumée et qui sont répartie en fonction de leurs besoins

```
1: entier indiceListePRGPU1 = 0 ;
2: entier indiceListePRGPU2 = 0 ;
3: entier indiceListePRCPU1 = 0 ;
4: entier indiceListePRCPU2 = 0 ;
5: entier NombreTachesListePRGPU1Machinej = 0 ;
6: entier NombreTachesListePRGPU2Machinej = 0 ;
7: entier NombreTachesListePRCPU1Machinej = 0 ;
8: entier NombreTachesListePRCPU2Machinej = 0 ;
9: pour tache allant de 0 à NombreTachesPr faire
10:     si ng(ListeTachesPr[tache].IndiceVM) ≠ 0 alors
11:         si nh(ListeTachesPr[tache].IndiceVM)>nr(ListeTachesPr[tache].IndiceVM) alors
12:             ListePRGPU1Machinej[indiceListePRGPU1].IndiceVM = ListeTachesPr[tache].IndiceVM ;
13:             indiceListePRGPU1 = indiceListePRGPU1 + 1 ;
14:             NombreTachesListePRGPU1Machinej++ ;
15:         sinon
16:             ListePRGPU2Machinej[indiceListePRGPU2].IndiceVM = ListeTachesPr[tache].IndiceVM ;
17:             indiceListePRGPU2 = indiceListePRGPU2 + 1 ;
18:             NombreTachesListePRGPU2Machinej++ ;
19:         fin si
20:     sinon
21:         si ng(ListeTachesPr[tache].IndiceVM) = 0 alors
22:             si nh(ListeTachesPr[tache].IndiceVM)>nr(ListeTachesPr[tache].IndiceVM) alors
23:                 ListePRCPU1Machinej[indiceListePRCPU1].IndiceVM = ListeTachesPr[tache].IndiceVM ;
24:                 indiceListePRCPU1 = indiceListePRCPU1 + 1 ;
25:                 NombreTachesListePRCPU1Machinej++ ;
26:             sinon
27:                 ListePRCPU2Machinej[indiceListePRCPU2].IndiceVM = ListeTachesPr[tache].IndiceVM ;
28:                 indiceListePRCPU2 = indiceListePRCPU2 + 1 ;
29:                 NombreTachesListePRCPU2Machinej++ ;
30:             fin si
31:         fin si
32:     fin si
33: fin pour
```

4. Implémentation de l'heuristique

4.1 Modélisation du programme

Dans le cadre de ce projet une approche modulaire a été utilisée pour réaliser le programme, de plus ce dernier a été réalisé en langage c. Comme l'approche modulaire est proche de l'approche sous forme de classe, la structure du programme sera représentée sous forme de d'un diagramme ressemblant à un diagramme de classe. Voici donc la représentation de la structure du programme.

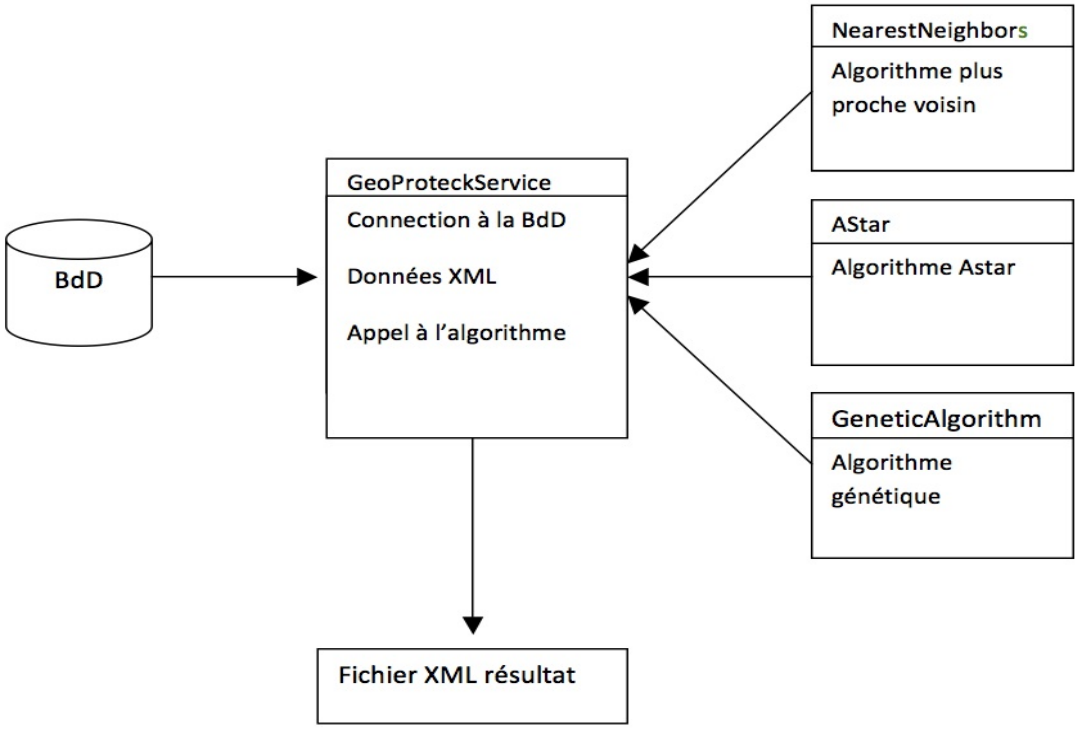

FIGURE 4.1 – Représentation de la structure du programme

Comme on peut voir sur la représentation, le programme est composé de trois parties. La première partie du programme correspond à la gestion des données issues du fichier de données qui a été réalisé par Vincent T'Kindt pour l'implémentation du modèle mathématique à l'aide du solveur CPLEX. La deuxième partie du programme correspond à mon implémentation de l'heuristique présenté dans le chapitre précédent. La dernière partie correspond au programme principal, dans lequel les différentes fonctions issues des deux parties précédentes sont appelés.

Je vais présenter dans ce qui suit l'ensemble de ces parties.

4.2 Gestion des données

Pour la gestion des données, une structure composée de plusieurs tableaux multidimensionnels permettant de stocker de façon logique les différentes données du problème. Dans le début du fichier data.h on retrouve la définition de valeur max suivante :

```
#define MAXVALUE 1000000000.0 //permet de définir une valeur maximale
#define MaxTasks 500 //permet de définir le nombre de tâche maximum
#define MaxMachines 10 //permet de définir le nombre de machine maximum
#define MaxEdges 1024 //permet de définir le nombre de arc maximum
#define MaxTimeHorizon 30 //permet de définir la valeur maximal de l'horizon de planification
```

On retrouve ensuite dans ce même fichier la définition des structures. La première structure définie est celle pour stocker les tâches. Pour chaque tâche ces informations sont si elle préemptable ou non, son tableau des $u_{i,t}$, ses besoins en ressources, ses affinités et la liste des machines sur lesquelle elle peut aller.

```
typedef struct{
short int QtyCPU;
short int QtyGPU;
short int QtyRAM;
short int QtyHDD;
short int isPreemptable; // 1 si la tâche est préemptable, 0 sinon
short int CostPreemption;
short int LIsToBeProcessed[MaxTimeHorizon]; // tableau des (uit)
short int LAffinities[MaxTasks]; // tableau des affinité pour la tâche i (aij)
short int LPreAssignement[MaxMachines]; // tableau de préaffectation(qik)
}SDEFTask;
```

La structure suivante permet de stocker les informations pour chaque serveur. Les informations qu'on retrouve dans cette structure pour une machine j sont : la quantité disponible pour chaque caractéristique physique ainsi que leurs coûts d'utilisation respectif et aussi le vitesse de chargement du contexte.

```
typedef struct{ // Data structure associated to the machines
short int CostCPU;
short int CostGPU;
short int CostRAM;
short int CostHDD;
short int QtyCPU;
short int QtyGPU;
short int QtyRAM;
short int QtyHDD;
short int SpeedContext;
}SDEFMachines;
```

La structure suivante permet de stocker les informations du réseau. Les informations qu'on retrouve dans cette structure sont le nombre d'arc sur le réseau. La bande passante requise pour la migration de tâchess la bande passante requise pour la communication entre deux tâches. On trouve aussi le tableau avec pour chaque arc le nombre de couple de machine sur cet arc et le tableau suivant est le détail pour chaque arc des couples.

```
\begin{verbatim}
typedef struct{ // Data structure associated to the network
short int NbEdges; // The number of links in the network
short int MigrateBandwidth; // The bandwidth reserved for a job migration (bii)
short int MaxBandwidth; // The maximum bandwidth of any link (b)
short int ComBandwidth[MaxTasks][MaxTasks]; // data bij
short int NbMachinesByEdge[MaxEdges]; // The number of couples of machines (j,j') usigne the l
short int ListOfMachinesByEdge[MaxEdges][MaxMachines*MaxMachines][2]; // The Pll'
}SDEFNetwork;
```

La dernière structure est la plus importante, c'est celle qui est utilisée par la suite comme elle regroupe l'ensemble des structures précédentes plus d'autres informations comme l'horizon de planification, le nombre de tâches et le nombre de machines, ainsi que le tableau des coûts unitaires d'allumage d'un serveur.

```
typedef struct {
                int TimeHorizon; //horizon de planification
int NbTasks; //nombre de tâches à planifier
int NbMachines; //nombre de machines disponible
SDEFTask ListOfTasks[MaxTasks]; //tableau pour les tâches, chaque tâche possède sa propre stru
SDEFMachines ListOfMachines[MaxMachines]; //tableau pour les machines, chaque machine possède
SDEFNetwork Network; //structure du réseau
short int CostTurnOn[MaxTimeHorizon]; // tableau des beta(t)
} SDEFData;
```

Dans le fichier data.cpp on retrouve la fonction GetData() qui permet de lire le fichier de données et de remplir la structure de données. Ce fichier contient aussi les fonctions permettant d'accéder aux données contenue dans la structure DATA.

Les premières fonctions sont celles permettant d'accéder au données générale du problème :
- int T() : permet d'avoir accès à la valeur de l'horizon de planification.
- int N() : permet d'avoir accès à la valeur du nombre de tâches à planifier.
- int M() : permet d'avoir accès à la valeur du nombre de machines disponible.

Les fonctions qui suivent permettent d'accéder au données liées aux tâches du problème :
- short int nc(unsigned int i) : permet d'avoir accès à la valeur de CPU requise pour exécuter la tâche i.
- short int ng(unsigned int i) : permet d'avoir accès à la valeur de GPU requise pour exécuter la tâche i.
- short int nr(unsigned int i) : permet d'avoir accès à la valeur de RAM requise pour exécuter la tâche i.
- short int nh(unsigned int i) : permet d'avoir accès à la valeur de HDD requise pour exécuter la tâche i.
- short int u(unsigned int i,unsigned int t) : permet de savoir si la tâche i doit être exécutée à l'instant t
- short int a(unsigned int i,unsigned int k) : permet de savoir si il existe une affinité entre la tâche i et la tâche k.
- short int q(unsigned int i,unsigned int j) : permet de savoir si la tâche i peut être affecté sur la machine j.

- short int b(unsigned int i,unsigned int j) : permet de connaître la bande passante requise pour que la tâche i communique avec la tâche j.
- short int R(unsigned int i) : permet de savoir si la tâche i est préemptable ou non.
- short int rho(unsiqned int i) : permet de connaitre la pénalité de non exécution de la tâche i.
- short int mt(unsigend int i) : permet de connaitre la durée de migration de la tâche i.
- short int rt(unsigend int i,unsigned int j) : permet de connaitre le temps de resume de la tâche i sur la machine i.

Pour accéder au différentes information liés au machines, on a les fonctions suivante :

- short int mc(unsigned int j) : retourne la charge maximale de CPU accepté par la machine j.
- short int mg(unsigned int j) : retourne la charge maximale de GPU accepté par la machine j.
- short int mr(unsigned int j) : retourne la charge maximale de RAM accepté par la machine j.
- short int mh(unsigned int j) : retourne la charge maximale de HDD accepté par la machine j.
- short int alphac(unsigned int j) : retourne le coût d'utilisation du CPU de la machine j.
- short int alphag(unsigned int j) : retourne le coût d'utilisation du GPU de la machine j.
- short int alphar(unsigned int j) : retourne le coût d'utilisation de la RAM de la machine j.
- short int alphah(unsigned int j) : retourne le coût d'utilisation de la HDD de la machine j.
- short int beta(unsigned int t) : retourne le coût unitaire d'allumer une machine à l'instant t
- short int v(unsigned int j) : retourne le vitesse de chargement du contexte de la machine j.

Afin d'accéder au données liées au réseau, on utilise les fonctions suivante :

- short int maxb() : retourne la valeur du maximum de bande passante pour tout les arcs.
- short int NbEdges() : retourne le nombre de d'arcs dans le réseau.
- short int NbMachEdge(unsigned int e) : retourne le nombre de couples de machines qui utilise l'arc e.
- void CoupleMachines(unsigned int e, unsigned int pos, unsigned int &Mach1, unsigned int &Mach2) : permet d'obtenir la position de la Mach1 et la Mach2 sur l'arc e.

En plus de toutes ces fonctions, il existe une fonction nommée DisplayData() qui permet d'afficher dans la console l'ensemble des données de la structure DATA.

4.3 Implémentation de l'heuristique

Après avoir présenté l'existant concernant la gestion des données, je vais dans cette partie présenter la manière dont j'ai implémenté mon heuristique. Pour implémenter mon heuristique, j'ai crée un certain nombre de structures permettant d'effectuer les différentes opérations nécessaire pour construire l'ordonnancement. Cette structure se trouve dans le fichier Traitement.h et sera utilisée pour les fonctions du fichier Traitement.cpp. Je vais, dans un premier temps, présenter le contenu du fichier Traitement.h pour ensuite présenter les différentes fonctions de Traitement.cpp dont les algorithmes sont présentés dans le chapitre précédent.

4.3.1 Traitement.h

Afin de pouvoir manipuler les données sans écraser les données de bases contenues dans la structure DATA, j'ai crée une série de structure afin d'avoir une copie des données sur lesquelles les opérations seront effectuées.

Les premières données définies dans ce fichier sont les mêmes que celles du fichier data.h afin que les dimensions des tableaux soient cohérentes.

```
#define MAXVALUE 1000000000.0
#define MaxTasks 500
```

```
#define MaxMachines 10
#define MaxEdges 1024
#define MaxTimeHorizon 30
```

J'ai ensuite redéfini une structure serveur permettant de stocker pour chaque machine physique, l'indice du serveur, le coût normalisé associé ainsi qu'un booléen permettant de définir si le serveur est allumé ou non.

```
typedef struct{ //Structure de données pour contenir le numéro des machines avec leur cout nor
int IndiceServeur;
float CoutNorm;
bool ON;

}Serveur;
```

La deuxième structure définie est celle permettant de stocker les valeurs des caractéristiques de chaque machine physique afin de pouvoir calculer et stocker au fur et a mesure des affectations des tâches, les caractéristiques résiduelles

```
typedef struct{
int CPU;
int GPU;
int HDD;
int RAM;
}CaractServeur;
```

La structure suivante permet d'avoir la liste uniquement des serveurs qui allumés.

```
typedef struct{ //Structure de données pour contenir le numéro des machines avec leur cout nor
int IndiceServeur;
float CoutNorm;

}ServeurON;
```

Cette structure permet de stocker les valeurs de bande passante entre deux machines

```
typedef struct{
int Mach1;
int Mach2;
int BdePassanteDispo;

}Reseau;
```

Les structures suivantes permettent de stocker les listes contenant les tâches.

```
typedef struct{ //Structure de données permettant de stocker les indice des VMs tq besoins en
short int IndiceVM;
signed int prio;
}HDDRAM;

typedef struct{ //Structure de données permettant de stocker les indice des VMs tq besoins en
```

```
short int IndiceVM;
signed int prio;
}RAMHDD;

typedef struct{
short int IndiceVM;
int prio;
}Pream;
```

Cette structure permet de stocker les intervalles sur lesquels travaille l'heuristique, on stocke dans cette dernière, pour chaque intervalle calculé, son indice, sa borne inférieure et sa borne supérieure.

```
typedef struct{
short int IndiceInter;
short int BorneInf;
short int BorneSup;
}Interval;
```

La structure suivante permet de stocker la valeur de cout d'utilisation d'une VM i sur une machine j allumée.

```
typedef struct{
short int indiceVM;
float cout;

}ctij;
```

La structure suivante permet de stocker la valeur de cout d'utilisation d'une VM i sur une machine j non allumée

```
typedef struct{ //
short int indicePM;
float gain;
}gaj;
```

Cette structure va permettre de stocker les informations concernant l'ordonnancement de l'ensemble des tâches sur les machines.

```
typedef struct{
short int IndiceTache;
short int IndiceMachine;
bool affecter;
int dureeExe; //permet de stocker le temps d'execution réel de la tache IndiceVM afin de respe
int dureeSus; //permet de stocker le temps durant lequel la tâche i a été suspendu
}Ordo;
```

La dernière structure est la structure principale, nomée composé des structures précédentes plus d'un certain nombre de valeur permettant de stocker : le nombre de tâches pour chaque liste, le nombre d'intervalles, le nombre de serveurs allumés.

```
typedef struct{
int NbPr;
int NbHDDRAMGPU;
int NbHDDRAMGPUPr;
int NbHDDRAMGPUMachinej;
int NbRAMHDDGPUMachinej;
int NbHDDRAMCPUMachinej;
int NbRAMHDDCPUMachinej;
int NbHDDRAMCPU;
int NbHDDRAMCPUPr;
int NbRAMHDDGPU;
int NbRAMHDDGPUPr;
int NbRAMHDDCPU;
int NbRAMHDDCPUPr;
int NbInterval; //Permet de stocker le nombre d'intervalle qui découpe l'horizon de planificat
int NbServeurOn; //Permet de stocker le nombre de machine physique allumé
Serveur ListOfServer[MaxMachines];
CaractServeur ListOfServeurbis[MaxTimeHorizon][MaxMachines];
Reseau ListOfReseau[MaxTimeHorizon][MaxEdges][MaxMachines];
ServeurON ListOfServerOn[MaxMachines];
Interval ListOfIntervalles[MaxTimeHorizon];
Pream ListofTasksPr[MaxTasks];
//Listes des taches non préamtable GPU
HDDRAM ListOfTasks1GPU[MaxTasks];
RAMHDD ListOfTasks2GPU[MaxTasks];
//Listes des taches non préamtable CPU
HDDRAM ListOfTasks1CPU[MaxTasks];
RAMHDD ListOfTasks2CPU[MaxTasks];
//Liste des taches préantable GPU
HDDRAM ListOfTasks1GPUPr[MaxTasks];
HDDRAM ListOfTasks1GPUMachinej[MaxTasks];
RAMHDD ListOfTasks2GPUPr[MaxTasks];
RAMHDD ListOfTasks2GPUMachinej[MaxTasks];
//Liste des taches préantable CPU
HDDRAM ListOfTasks1CPUPr[MaxTasks];
HDDRAM ListOfTasks1CPUMachinej[MaxTasks];
RAMHDD ListOfTasks2CPUPr[MaxTasks];
RAMHDD ListOfTasks2CPUMachinej[MaxTasks];
ctij ListeOfctij[MaxTasks][MaxMachines];
gaj ListeOfgaj[MaxMachines];
Ordo  ListOfOrdo[MaxTimeHorizon][MaxTasks];
}Trait;
```

4.3.2 Traitement.cpp

Dans le fichier Traitement.cpp on retrouve toutes les fonctions dont les algorithmes ont été présenté précédemment. Je vais juste maintenant donner les noms des fonctions qui sont implémentées.

– void CalculInterval() : elle permet de calculer l'ensemble des intervalles et de remplir la structure prévus pour stocker les informations liés aux intervalles 1.

- void CalculCoutNorm() : elle permet de calculer pour l'ensemble des machines les coûts normalisés de chaque machine, de remplir la structure prévue à cet effet et de la classer par coûts normalisés croissant. 2.
- void ConstructionListesTachesNonPr(unsigned int i) : elle permet de remplir les quatre listes des tâches non préemptables pour l'intervalle i placé en paramètre.5.
- void ConstructionListesTachesPr(unsigned int i) : cette fonction permet de réaliser le même travail que la précédente fonction mais pour les tâches préemptables.
- void ConstructionListeTachesPr(unsigned int i) : cette fonction permet de remplir la liste des tâches préemptables sans les séparer entre les différents besoins sur l'intervalle i placé en paramètre.
- void Ordonnancement(unsigned int i) : ces la fonction principale de l'heuristique, elle permet de faire appel a l'ensemble des fonctions qui permettent de construire l'ordonnancement sur l'intervalle i 9.
- int CalculTotalCost() : c'est la dernière fonction qui appelée dans l'heuristique car elle permet de calculer la valeur de la fonction objectif de l'ordonnancement déterminé.

On va maintenant détailler l'ensemble des fonctions qui sont appelées lors de l'exécution de la fonction Ordonnancement.

- void CalculPrioGPU(unsigned int indiceServeur,unsigned int indice) : c'est la première fonction que l'on appelle pour construire l'ordonnancement, elle permet de calculer les priorités pour les deux listes des tâches non préemptables qui ont besoins de GPU sur l'intervalle indice et avec la machine indiceServeur 8.
- void OrdoGPU (unsigned int indice,unsigned int indiceServeur) : cette fonction permet de faire l'affectation des tâches sur la machine indiceServeur sur l'intervalle indice 13. Au cours de son exécution cette fonction utilise les fonctions suivantes :
  - int CalculFesabiliteReseau(unsigned int indiceVM1,unsigned int indiceVM2,unsigned int indiceServeurVM2,unsigned int indiceServeur,unsigned int indice) : c'est la fonction qui permet dans le cas où une tâche que l'on veut affecter a une affinité avec une autre tâche déjà affecté sur une autre machine de vérifier que le réseau permet la communication entre ces deux machines.
  - void MaJReseau(unsigned int indiceVM1,unsigned int indiceVM2,unsigned int indiceServeurVM2,unsigned int indiceServeur,unsigned int indice) : fonction qui dans le cas où la fesabilité du réseau permet de mettre la VM1 sur le serveur indiceServeur permet de mettre a jour l'utilisation de la bande passante.
- void CalculPrioCPU(unsigned int indiceServeur,unsigned int indice) : c'est la fonction que l'on appelle pour construire l'ordonnancement, elle permet de calculer les priorités pour les deux listes des tâches non préemptables qui ont besoins uniquement de CPU sur l'intervalle indice et avec la machine indiceServeur 8.
- void OrdoCPU(unsigned int indice,unsigned int indiceServeur) :c'est la même fonction que pour les tâches avec des besoins en GPU, ici on travaille que sur les listes des tâches avec des besoins CPU. Elle fait appel au même fonction pour gérer le réseau que OrdoGPU.
- void OrdoTachePreSurServeurOn(unsigned int i) : cette fonction permet après affecté les tâches non préemptables, de passer à l'affectation des tâches préemptables sur l'intervalle i.

Je vais maintenant détailler les fonctions qui sont appelées dans OrdoTachePreSurServeurOn(unsigned int i) :

- void OrdoGPUPr(unsigned int i, unsigned int indiceServeurON)(resp. void OrdoCPUPr(unsigned int i, unsigned int indiceServeurON)) : ces fonction permet de déterminer la liste des tâches préemptables ayant des besoins en termes de GPU(resp. CPU) dont l'indice du serveur allumé placé en paramètre correspond à l'indice de la machine pour laquelle le cout d'affectation est minimum. Elle fait appel après aux fonctions suivantes :

  - void CalculPrioGPUPr(unsigned int indiceServeurON,unsigned int i) (resp. void CalculPrioGPUPr(unsigned int indiceServeurON,unsigned int i)) : fonctions permettant de calculer les priorités et de trier les listes des tâches allant sur le machine indiceServeurON par ordre décroissants des priorités.
  - void AffectationGPUPre(unsigned int i, unsigned int indiceServeurON) (resp. void AffectationGPUPre(unsigned int i, unsigned int indiceServeurON)) : ces fonction est la même que OrdoGPU (resp. OrdoCPU) mais pour les tâches préemptables. Elles font appel aux même fonctions que OrdoGPU (resp. OrdoCPU).
- void AllumageMachine(unsigned int i) : permet de déterminer si sur l'intervalle i il faut rallumer une machine pour affecter des tâches préemptable non affectées. Si il faut allumer une nouvelle machine, elle fera appel aux fonctions AffectationGPUPre et AffectationCPUPre

5. Résultats

5.1 Mise en place des tests

Afin de tester les résultats obtenus à l'aide de l'heuristique, je suis reparti du testeur réalisé par Vincent T'kindt pour tester l'implémentation du modèle mathématique avec CPLEX. Ce testeur contient un générateur d'instance, qui permet de générer 20 instances pour 6 scénarios différents, on entend par scénario que toutes les instances d'un scénario possèdent le même nombre de tâches et de machines, seul les autres données varient.

L'objectif du testeur que je devais réaliser pour tester mon heuristique, était pour l'ensemble des instances de chaque scénario de lancer les programme de la méthode exact et celui de l'heuristique. Un fois les programme exécuté, récupérer les données des deux programmes qui sont écrites dans des fichiers textes après leurs exécutions afin de comparer les résultats. Afin de comparer les résultats, je me suis servie de la déviation des fonctions objectifs entre les deux méthodes grâce à la formule de l'écart type **??**. La sortie du testeur est alors un fichier de statistique regroupant par scénario les données des 20 instances.Parmis ces données on retrouve les nombre d'instances résolues et non résolues pour l'heuristique et la méthode exacte, le temps d'exécution min,moyen et max pour les deux méthodes ainsi que la déviation min,moyenne et max.

5.2 Analyse des résultats

Après une première exécution du programme sur le scénario 1, sur les 20 instances mon heuristique retournée que 8 instances résolues alors que la méthode exacte trouvée 18 instances réalisables.

Ce premier résultat m'a permis de remettre en cause certaines choses de mon heuristique. Le problème qui a été résolu était la détection des ordonnancements réalisables ou non. Suite à la modification de la fonction en question on obtenait 17 instances résolues ce qui était déjà mieux.

Après ce deuxième résultat, l'analyse de la valeur de la déviation moyenne qui était de plus de 89% faisait ressortir que la gestion des tâches préemptables était peut être à revoir ainsi que la fonction de calcul de la fonction objectif. Suite à la correction de ces problèmes, la déviation moyenne était de 6%.

Voici les résultats obtenus suite à la campagne de test réalisée sur les cinq premiers scénarios.

		Heuristique					Méthode exacte							
Scénario	N & M	# Non Résolues	# Résolues	Tps Min	Tps Moyen	Tps Max	# Non Résolues	# Résolues	Tps Min	Tps Moyen	Tps Max	Déviation Min	Déviation Moy	Déviation Max
Sc1	8 & 2	4	16	0	0,14	1	2	18	0	0,71	1	2,65	5,55	18,38
Sc2	11 & 3	12	8	0	0	0	9	11	0	1,9	4	6,97	8,82	19,42
Sc3	15 & 4	12	8	0	0	0	1	19	1	11,32	56	4,26	11,23	21,28
Sc4	18 & 5	18	2	0	0	0	0	20	9	108,25	508	2,08	6,65	11,23
Sc5	21 & 5	19	1	0	0	0	3	8	2	425	1694	12,23	12,23	12,23

FIGURE 5.1 – Tableau de résultat sur 5 scénarios

6. Déroulement du projet

Dans ce chapitre je vais faire la synthèse du déroulement du projet, j'évoquerai aussi les difficultés rencontrées dans la réalisation de ce dernier.

6.1 Déroulement du projet

Lors de la réalisation du cahier de spécification, j'avais établi un diagramme de Gantt 6.1 à partir des informations de l'époque.

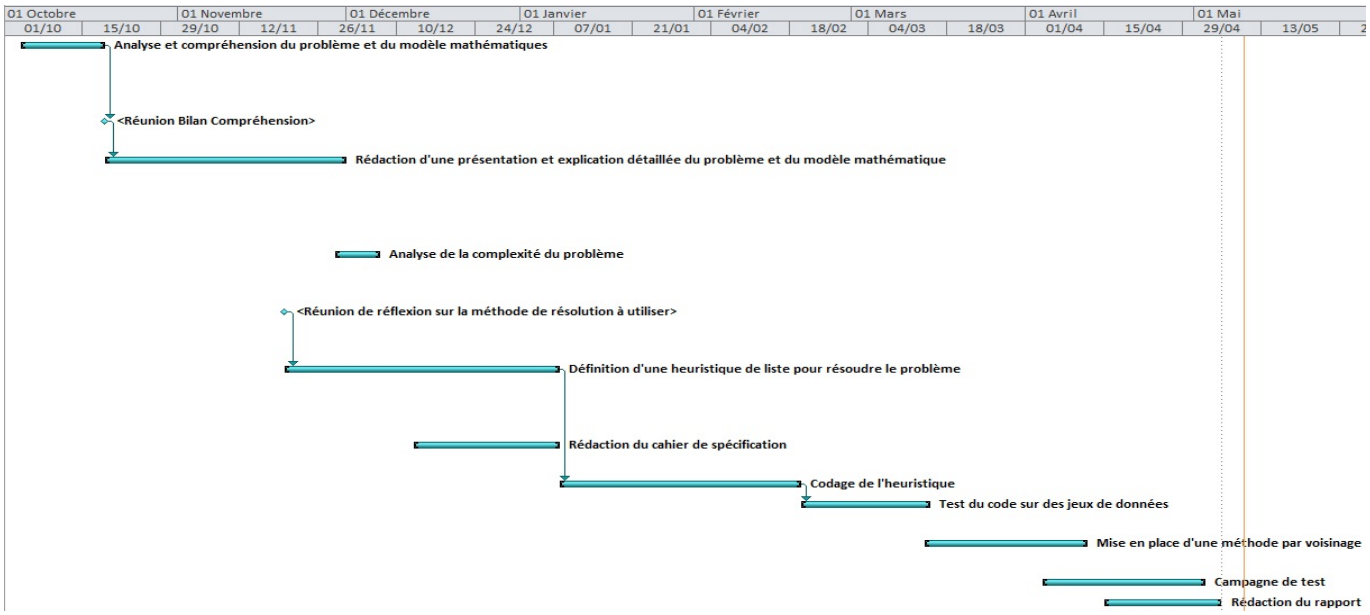

FIGURE 6.1 – Diagramme de Gantt prévisionnel

Cependant le planning réel du projet a été le suivant :

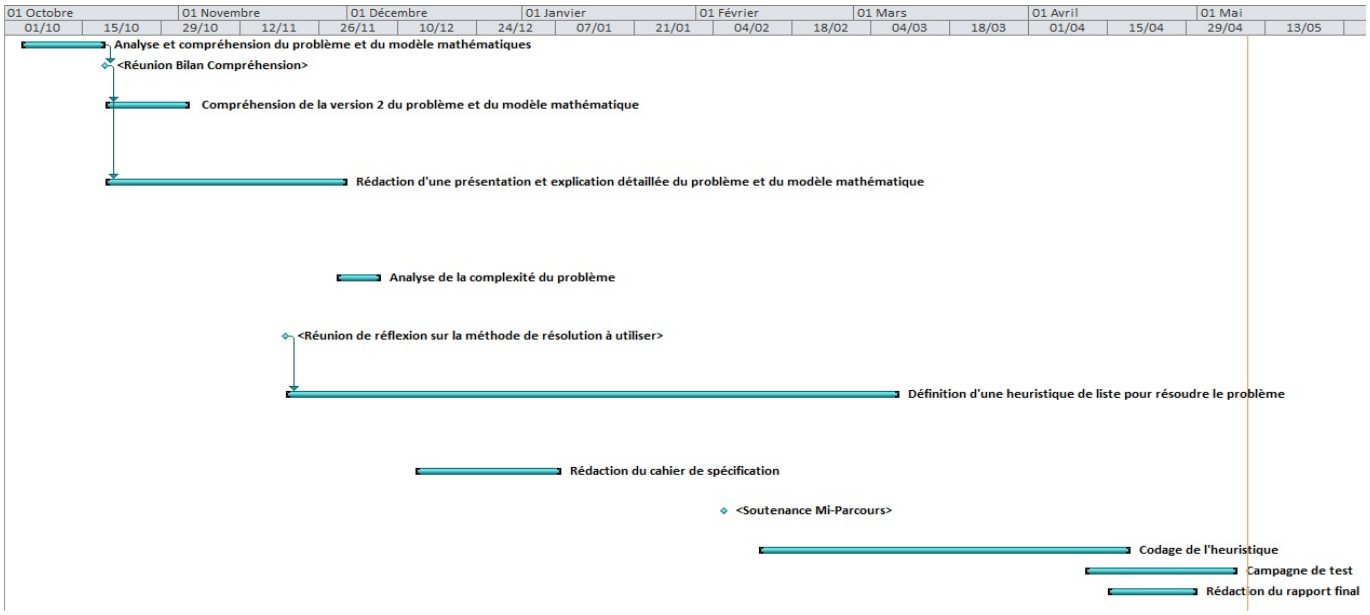

FIGURE 6.2 – Diagramme de Gantt

On remarque des différences entre les deux diagrammes de Gantt, la première différence est que le temps pour concevoir l'heuristique de liste à été plus longue que prévue. Cela est du en partie à la difficulté du problème. La schématisation du problème en sous problème de bin-packing contraint était bien et rapide à trouver mais trouver une méthode pour résoudre le sous problème a pris du temps. De plus, la tâche qui constitait à réaliser une résolution par voisinage a du être mise de coté car l'heuristique de liste a été plus longue à réaliser. Par ailleurs, dans le précédent planning il y a avait deux tâches qui consistaient à réaliser des tests. Je les ai regroupées en seule tâche qui est celle de la campagne de test. Puisque des tests ont été faits au fur et a mesure de la programmation de l'heuristique, ont été effectués sur des instances issues de la campagne de tests.

6.2 Difficultés rencontrées

Lors de ce projet, j'ai rencontré quelque difficultés que je vais décrire dans cette partie.

La plus grande difficulté rencontré au cours de se projet, a été l'élaboration de l'heuristique de liste pour résoudre le problème. Comme le problème est un problème NP-complet au sens fort et qu'il y a de nombreuses contraintes qui devaient être prise en compte, cela a compliqué la mise en oeuvre de l'heuristique.

La deuxième difficulté rencontré a été au moment de lancer la campagne de test. Comme je devais utilisé l'implémentation de la méthode exact réalisé par mon encadrant pour comparer les résultats obtenu par les deux méthodes. Le problème qui est survenu était que l'implémentation réalisé par Vincent T'kindt du modèle mathématique avec CPLEX était configuré pour la configuration de son ordinateur qui possédé une plus grande capacité au niveau processeur et mémoire vive.

Conclusion

Ce projet de fin d'étude m'a beaucoup appris. En effet, c'est la première fois sur mes 3 années de formation que je cherche à mettre en oeuvre une heuristique et à l'implémenter pour un problème concret. Cela m'a permis de mettre en pratique un certain nombre de concept mathématiques et informatiques pour résoudre des problèmes vus au cours de ces 3 années, particulièrement ceux liés à la Recherche Opérationnelle.

Ce projet m'a permis aussi de mettre en oeuvre mes connaissances en termes de modélisation, conduite de projet et programmation informatique.

Par ailleurs, grâce à ce projet, j'ai également acquis de nouvelles connaissances sur la mise en place d'une campagne de test dans le but de vérifier des résultats obtenus à partir d'une méthode approchée à un problème par rapport à des résultats obtenus par une méthode exacte.

Annexes

Résolution d'un problème off-line de consolidation de serveurs

Département Informatique
5e année
2012 - 2013

Rapport PFE

Résumé : Ce rapport a été écrit à la suite d'un projet de fin d'études à l'école d'ingénieurs Polytech'Tours, département informatique. Principalement basé sur la réalisation d'une heurtistique afin de résoudre un problème NP-complet. Ce rapport reprend les différentes étapes qui ont permis de déboucher sur l'implémentation d'une heuristique qui permet des résultats.

Mots clefs : Heuristique de liste, Problème NP-Complet, Bin-Packing contraint, Recherche Opérationnelle

Abstract: This report has been written following a graduation project at the engineering school of Polytech'Tours, IT section. Mainly based on the achievement of an heurtistique to solve an NP-complete problem. This report describes the various stages that helped lead to the implementation of a heuristic that allows the results.

Keywords: Heuristic list, NP-complete problem, Bin-packing constrain, Operational Research

Encadrant
Vincent T'Kindt
vincent.tkindt@univ-tours.fr

Étudiant
Cyrille PICARD
cyrille.picard@etu.univ-tours.fr

Université François-Rabelais, Tours