Collège Sciences et Technologies
UF Mathématiques et Interactions

Le problème de RCPSP

“Resource-Constrained Project Scheduling Models”

Joanne Affolter
Kea Horvath

CMI OPTIM L1
2019-2020

Introduction à l’ingénierie de l’optimisation

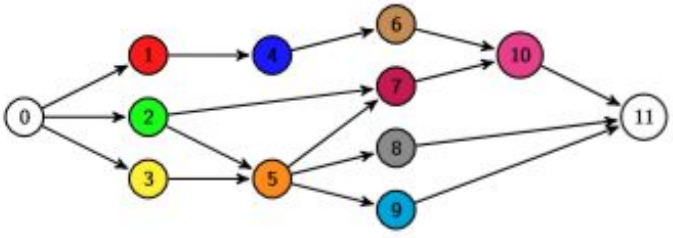
Table des matières

1. Introduction

Le problème de RCPSP - ou problème de gestion de projet à contraintes de ressources - est un problème d'optimisation assez répandu. Il consiste à ordonnancer un nombre fixé de tâches tout en minimisant la durée que l'on met à réaliser l'ensemble de celles-ci. Ce type de problème est apparu dans les années 50 en France. Il peut permettre de modéliser énormément de cas pratiques comme par exemple dans l'industrie (aéronautique, fabrication de matériaux), dans les hôpitaux (organisation des blocs opératoires, transport des produits), dans le transport ferroviaire, etc.

Plusieurs contraintes s'appliquent aux problèmes de RCPSP. Notre projet prendra en compte deux contraintes : une contrainte de précédence et une contrainte de ressources.
D'une part, il va falloir s'assurer que les activités figurant parmi la liste de précédence de l'activité à ordonnancer ont bien été placées. En effet, les différentes tâches sont liées et certaines d'entre elles ne pourront pas débuter avant que d'autres ne soient terminées.
D'autre part, il faudra vérifier qu'à chaque instant du projet, l'ensemble des ressources utilisées par les activités en cours ne dépasse par la limite maximale de consommation des ressources fixée au préalable. Dans la majeure partie des cas, plusieurs types de ressources différentes doivent être gérés en même temps. Il peut s'agir d'argent, de main d'oeuvre ou d'énergie. Dans notre projet, nous ne prendrons en compte qu'une seule ressource afin de faciliter la résolution du problème.
Notre objectif est donc de trouver le meilleur ordonnancement d'activités possible afin de terminer l'ensemble des tâches le plus vite possible, tout en respectant les contraintes fixées au préalable.

Voici un exemple de solution d'un problème simple où 10 activités sont à ordonnancer et où la quantité maximale de ressources à ne pas dépasser est égale à 4. i représente le nom de l'activité, p_i représente sa durée et b_i la quantité de ressources nécessaires pour réaliser la tâche i. On peut voir sur le graphe de gauche les précédences à respecter, et en dessous, une solution d'ordonnancement du problème, où la durée est représentée sur l'axe des abscisses et la consommation de ressources sur l'axe des ordonnées.



i	p_i	b_i
1	3	2
2	5	3
3	1	3
4	3	1
5	2	1
6	4	2
7	5	3
8	6	1
9	4	1
10	4	1

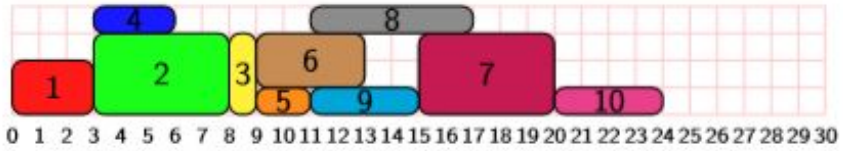


Figure 1 - Une solution d'un RCPSP à 10 tâches

Méthodes exactes pour le problème d'ordonnance de projet à moyens limités (RCPSP), Christian Artigues, CNRS & Université de Toulouse

Dans ce rapport, après avoir présenté la modélisation de notre problème, nous expliciterons plusieurs méthodes de résolution qui chercheront à se rapprocher le plus possible de l'optimal.

L'ensemble des algorithmes obtenus seront comparés expérimentalement entre eux, et avec un minorant, dont nous détaillerons le calcul.

Le rapport est organisé de la manière suivante. La section 2 présente le problème du RCPSP au centre de notre projet ainsi que sa modélisation. La section 3 présente des algorithmes de résolution heuristique de ce problème. La section 4 détaille le calcul du minorant. La section 5 présente les résultats des tests effectués sur le jeu de données fourni ainsi que leur analyse. La section 6 est dédiée à une conclusion sur le projet.

2. Formalisation du problème

Avant de mettre en place différentes stratégies de résolution, il est nécessaire de modéliser de manière claire et précise notre problème de RCPSP.

Données d'entrée et de sortie des algorithmes de résolution :

À chaque tâche est liée une durée, la quantité de ressources nécessaires pour réaliser cette tâche et des relations de précédence à respecter. Les principales données du problème à traiter, tirées du jeu de données à notre disposition, sont les suivantes.

- Les activités A_i à ordonnancer, chaque activité étant numérotée de 1 à n.
- La durée maximale C_{max} à ne pas dépasser
- La liste de précédence E_i associée à chaque activité : il s'agit de l'ensemble des activités devant être effectuées avant l'activité i. Dans l'exemple ci-dessous, l'activité 3 ne pourra pas débuter tant que l'activité 4 n'est pas terminée.
- La quantité de ressources maximale B_{max} à ne pas dépasser à chaque instant t.
- La durée p_i associée à chaque activité.
- La quantité b_i de ressources consommées par chaque activité.

```
jobs 4
horizon 30
PRECEDENCE:
1
2
3    4
4
RESOURCEAVAILABILITY 10
REQUESTS/DURATIONS:
 1 10 10
 2 5 5
 3 5 5
 4 6 5
```

Figure 2 - Capture d'écran d'un des fichiers texte appartenant au jeu de données mis à notre disposition

Pour chaque méthode de résolution, notre algorithme va prendre en entrée ces différentes données, et retourne la durée totale du projet, après ordonnancement des activités.

Représentation graphique des données du problème :

Nous pouvons représenter chaque activité A_i par un rectangle en 2 dimensions, dont la longueur dépend de sa durée p_i et la hauteur de la quantité b_i de ressources utilisées.

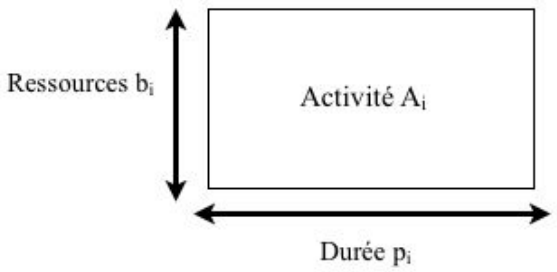


Figure 3 - Représentation graphique d'une activité A_i en fonction de sa durée et des ressources utilisées

La solution, quant à elle, peut être représentée graphiquement. La durée du projet est alors exprimée sur l'axe des abscisses et la quantité de ressources utilisées sur l'axe des ordonnées. La quantité de ressources maximale B_{max} à ne pas dépasser à chaque instant t peut être représentée par une droite de coefficient directeur $y = B_{max}$.

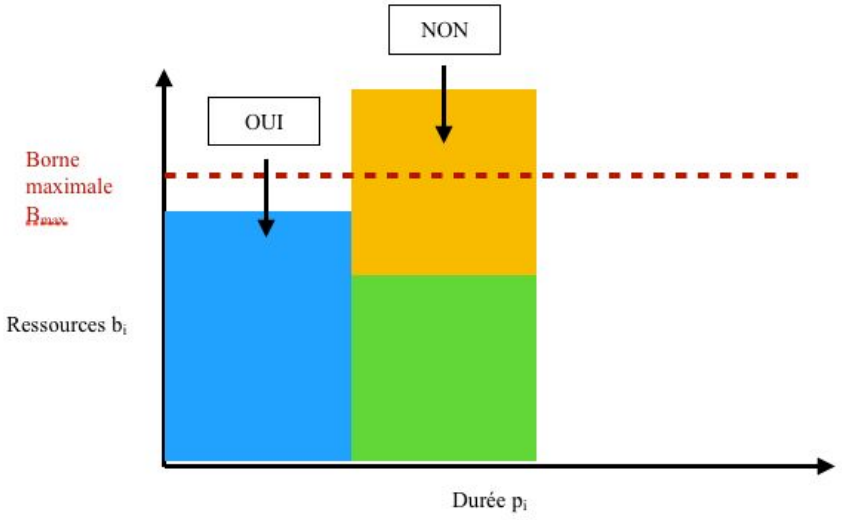


Figure 4 - Représentation graphique d'une solution au problème de RCPSP avec mise en avant du respect de la condition de ressources

Enfin, la relation de précédence peut être assimilée à un graphe : chaque sommet étant associé à une activité, et chaque arête à une relation de précédence. Reprenons l'exemple du jeu de données présenté plus haut. Le graphe associé aux précédences serait alors le suivant.

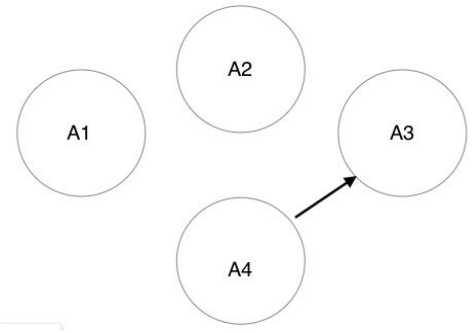


Figure 5 - Graphe de précédence

L'activité A4 doit être effectuée avant l'activité A3.
On représente cette relation de précédence par une arête du graphe.

Modélisation mathématique du problème :

L'objectif principal de notre problème est de minimiser la durée totale du problème. Il faut donc trouver la valeur la plus petite possible pour C_{max}, c'est-à-dire celle se rapprochant le plus du minorant dont nous détaillerons le calcul dans la partie 4. Le majorant de notre problème correspond à la valeur suivante de C_{max}. *Soient n le nombre d'activités à ordonnancer et p_i la durée de l'activité i, alors* $C_{max} = \sum_{i=1}^{n} p_i$.

Il s'agit de la somme de la durée de chaque activité du projet à ordonnancer. On suppose donc ici qu'aucune activité n'est effectuée en même temps.

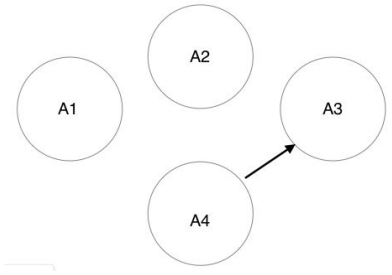
Les deux contraintes, quant à elles, peuvent se modéliser de la manière suivante.

Le respect de la contrainte de ressource s'exprime à travers la borne B_{max}. À chaque instant t, la somme des ressources utilisées par les activités en cours doit être inférieure à B_{max}, ce qu'on peut modéliser de la manière suivante.

Soient A_t les activités qui se déroulent à l'instant t et b_i la quantité de ressources consommées par l'activité i, alors $\forall t \in N$ *tel que* $t \in [0, C_{max}]$, $B_{max} \geq \sum_{Ai \in At} b_i$

Si cette inégalité est vérifiée, alors la contrainte de ressources est respectée.

Reprenons notre exemple pour illustrer la modélisation de la contrainte de précédence.



Si l'activité A_4 appartient à la liste de précédence de l'activité A_3, l'activité A_3 ne pourra pas commencer tant que l'activité A_4 ne sera terminée. On peut donc poser la relation suivante.

Soient S_i la date de début de l'activité i.
Alors il faut que : $S_3 > S_4 + p_4$

Pour revenir au cas général, on aura alors :
Soient E_i la liste de précédence de l'activité i, A_j les activités appartenant à cette liste, S_i le début de l'activité i et S_j le début d'une activité appartenant à E_i. Alors, $\forall A_j \in E_i, S_i > S_j + p_j$
Nous pourrons vérifier si ces deux contraintes sont respectées grâce au vérificateur de solutions, dont le contenu est détaillé en annexe. Cet algorithme prend en entrée une solution calculée par l'algorithme de résolution, et regarde pour chaque activité si les contraintes de précédence et de ressources sont bien vérifiées. Si tel est le cas, la solution est considérée comme réalisable.

3. Algorithmes de résolution

Dans cette section, nous présentons quatre méthodes de résolution heuristique du problème décrit dans la section 2. La première méthode, dite "absurde", consiste à parcourir la liste des activités classées selon leur numéro et de placer les activités dont les contraintes de précédence et de ressources sont vérifiées, et ce jusqu'à ce que toutes les activités soient placées. La seconde méthode de séparation de liste consiste à placer en priorité les activités de durée nulle ou dont la liste de précédence est vide. Pour cela, on sépare en amont les activités à ordonnancer en deux listes : une liste contenant les activités à placer en priorité et une liste constituée du reste des activités qu'on essaiera de placer dans un second temps. La troisième méthode est basée sur un tri de la liste des activités à placer en fonction de certains critères. Nous vous présenterons dans cette sous-section 4 algorithmes différents basés sur ce tri de liste mais prenant en compte des paramètres de tri différent. Suite à cela, nous vous présenterons un algorithme de recherche locale, qui prend le résultat obtenu par une des méthodes citées précédemment, et qui effectue des permutations d'activités pour tenter de trouver un meilleur résultat.

Ces 6 heuristiques sont basées sur le même algorithme qui permet de vérifier si les contraintes de précédence et de ressources sont vérifiées. Il s'agit du fichier "contraintes.py".

3.1. Présentation de l'algorithme "contraintes.py"

Penchons-nous dans un premier temps sur les conditions de vérification de la condition de précédence. Détaillons les différentes étapes à suivre pour vérifier si cette dernière est satisfaite.

Etape 1 - On vérifie si l'activité possède une liste de précédence

---

Algorithme 1

Entrée : numéro (ou indice) de l'activité à placer, liste de précédence de cette activité
Sortie : Une donnée caractérisant l'existence ou non d'une liste de précédence

```
Si longueur de la liste de précédence de l'activité i == 0   ← il n'y a pas de précédence à respecter
    retourner 1   ← on peut placer l'activité
Sinon
    retourner liste de précédence de l'activité i   ← les activités de cette liste doivent être placées avant
```

Si la liste de précédence est vide, la condition de précédence est satisfaite. Sinon, il faut vérifier que les activités de la liste de précédence ont bien été placées et on passe à l'étape 2.

Etape 2 - On vérifie si l'ensemble des activités de la liste de précédence ont déjà été placées

Algorithme 2
Entrée : numéro (ou indice) de l'activité à placer, liste de précédence de cette activité, liste des activités finies, la valeur retournée par l'algorithme 1
Sortie : Une donnée caractérisant la satisfaction ou non de la condition de précédence

```
res = 0  ← on initialise un compteur à 0
Si valeur retournée par l'algorithme 1 != 1   ← si il y a une condition de précédence à respecter
    Si longueur liste des activités finies == 0      ← si aucune activité n'a été placée
        retourner None  ← la condition de précédence n'est pas vérifiée
    Sinon
        Pour k dans liste de précédence de l'activité i
            Pour j dans liste des activités finies
                Si k == j
                    res = res + 1  ← l'activité k de la liste de précédence a déjà été placée, on ajoute 1 au compteur
            Si res == longueur liste de précédence   ← toutes les activités de précédence sont placées
                retourner 1    ← la condition de précédence est vérifiée
            Sinon
                retourner None  ← la condition de précédence n'est pas vérifiée
Sinon
    retourner 1  ← la condition de précédence est vérifiée car la liste de précédence est vide (étape 1)
```

Si chacune des activités de la liste de précédence de l'activité i a déjà été placée et figure dans la liste des activités terminées, la condition de précédence est vérifiée et notre programme retourne 1. On peut alors passer à l'étape 3 et déterminer à partir de quel instant on peut placer l'activité tout en respectant la contrainte de précédence. Si la condition de précédence n'est pas respectée, notre algorithme retourne None.

Etape 3 - On détermine à partir de quel instant l'activité peut débuter

Algorithme 3
Entrée : numéro (ou indice) de l'activité à placer, liste de précédence de cette activité, liste regroupant le temps où chaque activité placée se termine, la valeur retournée par l'algorithme 1
Sortie : Le temps à partir duquel l'activité peut être placée

```
Si longueur liste de précédence == 0    ← il n'y a pas de condition de précédence
    t = 0    ← on peut placer l'activité à partir du temps 0
    retourner t
Sinon
    t = 0    ← on initialise t à 0
    Pour j dans liste de précédence
        Si fin activité j > t    ← on regarde si les activités de la liste de précédence se terminent après t
            t = fin activité j    ← t correspond à l'instant où la dernière activité de précédence se termine
    retourner t    ← temps à partir duquel on peut placer l'activité en respectant la précédence
```

Notons que notre objectif est de chercher à toujours placer l'activité le plus tôt possible. Ainsi, pour les activités qui n'ont pas de condition de précédence à respecter et pour celles dont toutes les activités de précédence ont une durée nulle, on essaiera de les placer à partir du temps t = 0. Pour celles devant respecter une précédence, elles pourront être placées après que la dernière activité de la liste de précédence soit terminée, c'est-à-dire au temps t avec t = *max* (t_1, t_2, t_3..., t_n) *où* t_i *correspond à la* fin de l'activité i figurant dans la liste de précédence de l'activité à placer.

Après avoir vérifié la satisfaction de la contrainte de précédence, il est nécessaire de se pencher sur la vérification de la contrainte de consommation des ressources.
Nous avons décidé de créer dans un premier temps une liste qui va nous permettre de stocker la quantité de ressources disponibles à chaque instant du projet.

Etape 1 - Création d'une liste qui regroupe la quantité de ressources disponibles à chaque instant

---

Algorithme 4
Entrée : Bmax (quantité de ressources maximale à ne pas dépasser), Cmax (durée maximale du projet)
Sortie : liste des ressources disponibles à chaque instant

```
ressource_disponible = [ ]   ← on crée une liste vide
Pour i allant de 0 à Cmax    ← i parcourt chaque instant t du projet (i se déplace le long de l'axe des abscisses - voir Représentation graphique des données du problème - Section 2)
    ressource_disponible = ressource_disponible + [Bmax]   ← on initialise la liste avec Bmax
retourner liste ressource_disponible
```

D'un point de vue de la représentation graphique du problème, la liste « ressource_disponible » correspond à la place qu'il nous reste à chaque instant pour placer une activité. Ainsi, si cette liste contient une valeur nulle, cela revient à dire qu'aucune activité ne pourra être placée à cet instant. On en déduit facilement que si cette liste contient des valeurs négatives, alors la contrainte de ressources n'est pas respectée puisque la borne maximale B_{max} a été dépassée. La liste « ressource_disponible » va être primordiale pour la gestion des ressources au cours de la résolution du problème.

Reprenons notre exemple pour mettre en avant l'utilité de cette liste. Ici B_{max} = 10, et donc ressource_disponible = [10, 10, 10, …., 10].
Si nous cherchons à placer l'activité 4 (de durée p_4 = 5 et de consommation b_4 = 6) à partir du temps t = 0, alors on aura : ressource_disponible = [4, 4, 4, 4, 4, 10, 10, …., 10].
Si l'on veut placer l'activité 3 (de durée p_5 = 5 et de consommation b_5 = 5) à partir du temps t=0, cela est impossible car la quantité de ressources disponibles est égale à 4 (or 4 < 5). On pourra donc seulement placer l'activité 3 à partir du temps t = 5 pour respecter la contrainte de ressources.

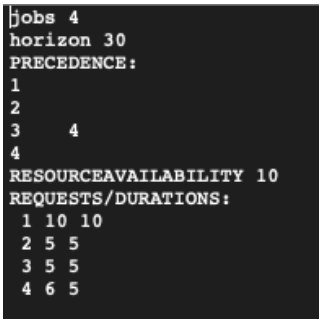


Figure 2 - Rappel du jeu de données de notre exemple

Déterminons maintenant à partir de quel instant l'activité peut être placée tout en respectant les contraintes de précédence et de consommation de ressources.

Etape 2 - On détermine à partir de quel moment on peut placer l'activité, tout en vérifiant si l'on dispose d'assez de ressources tout au long du déroulement de l'activité.

Algorithme 5

Entrée : numéro (ou indice) de l'activité à placer, durée de l'activité, consommation de ressources de l'activité, valeur retournée par l'algorithme 3 (temps à partir duquel la condition de précédence est vérifiée), valeur retournée par l'algorithme 4 (liste ressources disponibles), borne C_{max}
Sortie : début de l'activité

```
si durée activité == 0
    t = 0      ← on suppose que l'activité débute et se termine au temps t = 0
    retourner t
sinon
  pour k allant de t à Cmax    ← on se place à partir du temps où la condition de précédence est vérifiée
    pour j allant de t à t+durée activité    ← on vérifie si la condition est vérifiée durant toute l'activité
      si liste_ressource_disponible[j] - ressources[i-1] < 0    ← si il n'y a pas assez de place dans la liste
                                                                  ressources disponibles pour placer l'activité
        t = t + 1    ← on se place au temps suivant
    retourner t    ← t vérifie les contraintes de précédence et de ressources
```

Cet algorithme retourne donc la date de début de l'activité à placer, instant où les contraintes de ressources et de précédence sont vérifiées.

3.2. Heuristique idiote (Heuristique_1.py)

L'heuristique idiote consiste à placer les activités en suivant l'ordre croissant de leurs indices. On ne trie donc pas la liste regroupant les tâches à ordonnancer et on essaie de placer une activité après l'autre. Le seul but de cet algorithme est donc de vérifier si les contraintes de précédence et de ressources sont vérifiées pour chaque activité. Si tel est le cas, on peut placer l'activité, sinon on passe à l'activité suivante. On parcourt cette liste jusqu'à ce que toutes les activités soient placées.

Algorithme 6

Entrée : ensemble des données lues dans un des fichiers texte du jeu de donnée, listes regroupant les activités, le début de chaque activité, leur fin, l'ensemble des activités finies (liste vide au début), valeurs retournées par les algorithmes précédents

Sortie : date de fin du projet

```
state = "first"   ← state va nous permettre de vérifier si toutes les activités ont été placées

Si state == "first"   ← l'état "first" est l'état qui nous permet de placer les activités
    Pour i dans activites   ← on parcourt les activités en suivant l'ordre croissant de leurs numéros
        Si i != 0 et valeur retournée par l'algorithme 2 == 1 ← si l'activité n'a pas été placée et vérifie la
                                                                contrainte de précédence
            t2 = valeur retournée par l'algorithme 5  ← t2 correspond au début de l'activité
            debut_activite[i-1] = t2    ← on affecte à la liste des débuts d'activité la valeur t2
            Si duree[i-1] == 0    ← si l'activité à une durée nulle
                f = t2
            Sinon
                f = t2 + duree[i-1] ← f correspond a la fin de l'activité
            fin_activite[i-1] = f     ← on affecte à la liste des fins d'activité la fin de l'activité i
            Pour j allant de t2 à t2 + duree[i-1]    ← on modifie la liste ressources disponibles en enlevant la
                                                       les ressources utilisées par l'activité i à la quantité initiale
                liste_ressource_disponible[j] = liste_ressource_disponible[j] - ressources[i-1]
            activites_finies = activites_finies + [i]   ← on ajoute à la liste des activités finies l'activité i
            activites[i-1] = 0    ← on remplace le numéro de l'activité par un 0 pour montrer qu'elle a été placée
            state = "second"   ← on passe au 2e état
    Sinon   ← si l'activité a déjà été placée ou si la condition de précédence n'a pas été vérifiée
        state = "second"   ← on passe au deuxième état

Si state == "second"   ← cet état va nous permettre de vérifier si toutes les activités ont été placées
    Si longueur liste activités finies  != longueur liste activités   ← toutes les activités n'ont pas été placées
        state = "first"    ← on retourne au 1er état pour placer une nouvelle activité
    Sinon
        state = "end"   ← on passe à l'état final

Si state == "end"    ← cet état va nous permettre d'afficher les résultats
    fin = 0     ← on initialise la valeur de fin du projet à 0
    Pour k dans liste activités
        Si fin_activite[k-1] > fin   ← on parcourt la liste de fin des activités et on trouve la plus grande valeur
            fin = fin_activite[k-1]
    retourner fin   ← on retourne la date de fin du projet
```

Si on applique l'heuristique idiote à notre exemple, alors on place l'activité 1 du temps t = 0 jusqu'au temps t = 10. Toutes les ressources disponibles sont utilisées par cette activité. On aura alors : liste ressources_disponibles = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 10, ..., 10].
Puis on place l'activité 2 de t = 11 à t = 16. Sachant qu'elle consomme 5 quantités de ressources, on obtient: ressources_disponibles = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5, 10, 10, ... , 10].
On essaie de placer la 3e activité mais la condition de précédence n'est pas vérifiée donc on passe à la 4e activité.
On place l'activité 4 de t = 17 à t = 22. On ne peut pas la placer plus tôt car elle consomme 6 quantités de ressources, et qu'il y a seulement assez de ressources disponibles à partir de t = 17. La liste ressources disponibles devient donc ressources_disponibles = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5, 4, 4, 4, 4, 4, 10, 10,..., 10]
On peut finalement placer l'activité 3 de t = 23 à t = 28 et on obtient :
ressources_disponibles = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 10, 10, ... 10]
L'heuristique idiote nous retourne donc une date de fin du projet égale à 28 pour cet exemple. On voit bien que cet algorithme n'est pas optimal et qu'il est possible d'obtenir facilement une meilleure solution que celle-ci.

```
jobs 4
horizon 30
PRECEDENCE:
1
2
3    4
4
RESOURCEAVAILABILITY 10
REQUESTS/DURATIONS:
 1 10 10
 2 5 5
 3 5 5
 4 6 5
```

Figure 2 - Rappel du jeu de données de notre exemple

3.3. Heuristique de séparation de listes (Heuristique_2.py)

L'heuristique de séparation de listes consiste à séparer les activités en deux listes en fonction de plusieurs critères, dans le but de placer en priorité les activités qui nous posent le moins de problème. Ainsi, nous avons décidé de placer dans la première liste les activités qui n'ont aucune contrainte de précédence à vérifier ou qui ont une durée nulle. Le reste des activités est placé dans la seconde liste. Il suffit alors de placer dans un premier temps les activités de la première liste. Si elles ne peuvent pas être placées, on affecte ces activités à la deuxième liste et on essaiera de les placer par la suite. On rencontre cette situation lorsque les activités de la première liste ont une durée nulle mais possèdent une liste de précédence. Ensuite, nous pouvons nous concentrer sur les activités de la deuxième liste et les placer en utilisant la même méthode que pour l'heuristique absurde.

A première vue, cette méthode semble intéressante car elle permettrait d'augmenter les chances de satisfaire les conditions de précédence pour les activités de la seconde liste. En effet, par rapport à la méthode absurde, un plus grand nombre d'activités sera déjà placé lorsqu'on s'occupera des activités de la seconde liste possédant une précédence à respecter. D'ailleurs, placer dans un premier temps les activités de durée nulle semble pertinent car leur placement n'a aucun impact sur la consommation de ressources.

Voici l'algorithme qui permet de séparer les activités en deux listes distinctes.

Algorithme 7

Entrée : liste des activités, liste de la durée des activités, liste de précédence
Sortie : liste activites_sans_precedence (activités sans précédence ou de durée nulle), liste activites (activités restantes)

```
Si state == "pre first"     ← le state "pre first" permet de séparer la liste des activités en 2 listes
   activites_sans_precedence = [ ]   ← nouvelle liste pour  les activités de durée nulle ou sans précédence
   Pour i dans activites   ← on parcourt les activités par ordre croissant d'indices
        Si longueur liste precedence == 0 ou duree[i-1] == 0    ← on teste si la liste de précédence est
                                                                  vide ou si elles ont une durée nulle
            activites_sans_precedence = activites_sans_precedence + [i]   ← on l'ajoute a la nouvelle liste
            activites[i-1] = 0     ← on remplace son numéro par un 0 dans l'ancienne liste
        Sinon
            activites_sans_precedence = activites_sans_precedence + [0]  ← on remplace son numéro
                                                                           par un 0 dans la nouvelle liste
```

Après avoir utilisé cet algorithme, nous obtenons donc deux listes différentes. Nous plaçons ensuite les activités de la première liste, puis de la seconde, comme détaillé précédemment. Le placement des activités se fait de la même manière que pour l'heuristique absurde. C'est pourquoi nous avons choisi de ne pas détailler l'ensemble de l'algorithme et de résumer les différentes étapes de cette méthode dans le tableau suivant.

1e étape	Séparation de la liste des activités en 2 listes - listes activites_sans_precedence : activités de durée nulle ou précédence - liste activites : liste des activités privées de celles appartenant à la liste précédente
2e étape	Placement des activités de la liste activites_sans_precedence en s'appuyant sur l'heuristique absurde
3e étape	Placement des activités de la liste activites en s'appuyant sur l'heuristique absurde

Tableau résumant les différentes étapes de l'heuristique de tri de liste

Nous pouvons noter que cette heuristique nous renvoie le même résultat que la méthode absurde sur notre exemple. En effet, on obtient les deux listes suivantes suite à la première étape :
activites_sans_precedence = [1, 2, 0, 4], activites = [0, 0, 3, 0]
On placera ainsi les activités dans l'ordre suivant : [1, 2, 4, 3], qui est le même que celui de la méthode absurde.

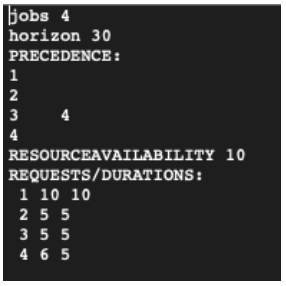


Figure 2 - Rappel du jeu de données de notre exemple

3.4. Heuristique de tri de liste (Heuristiques 3 à 6)

L'objectif principal des algorithmes de cette section est de trier la liste des activités avant de débuter leur placement. Nous avons décidé de baser ce tri sur deux critères différents : tout d'abord sur la quantité de ressources utilisées par chaque activité, puis sur la durée de celles-ci. Chacune des méthodes peut être déclinée de deux manières différentes car le tri peut se faire soit dans l'ordre croissant, soit dans l'ordre décroissant. Nous avons ainsi programmé 4 algorithmes reposant sur cette méthode :

- une heuristique de tri selon la consommation décroissante de ressources : Heuristique_3.py
- une heuristique de tri selon la consommation croissante de ressources : Heuristique_4.py
- une heuristique de tri selon la durée décroissante des activités : Heuristique_5.py
- une heuristique de tri selon la durée croissante des activités : Heuristique_6.py

Afin d'illustrer cette méthode, appuyons-nous sur un exemple de tri de liste par ordre décroissant de consommation de ressources.

Supposons que nous avons la liste suivante.

Numéro des activités	1	2	3	4	5
Quantité de ressources utilisées	5	7	3	8	10

Après le tri, nous obtenons cette liste et essaierons de placer les activités suivant ce nouvel ordre.

Numéro des activités	5	4	2	1	3
Quantité de ressources utilisées	10	8	7	5	3

A première vue, le tri par ordre décroissant semble le plus intéressant car nous avons vu avec les heuristiques précédentes que les activités les plus difficiles à placer vers la fin du problème étaient celles dont la consommation de ressources était élevée ou celles qui avaient une durée importante. Ainsi, les placer en premier nous permettrait de se défaire plus rapidement de ces activités.

Voici l'algorithme de base sur lequel se basent les 4 heuristiques de tri de liste. Ces 4 méthodes sont en effet très similaires et seules quelques paramètres vont différer d'un algorithme à l'autre. De manière évidente, l'algorithme de tri en fonction de la consommation des ressources prend comme paramètre de tri la liste "ressources" de notre problème, alors que celui en fonction de la durée utilise la liste "durée". Pour le tri croissant, on recherche l'élément dont le critère de tri est le plus petit, on l'affecte au minimum et on le place en premier dans la liste, et ce ainsi de suite jusqu'à ce que le tableau soit trié. Pour le tri décroissant, la méthode est la même sauf qu'on cherche le maximum. Ensuite, lorsque la liste des activités est triée, on peut placer les activités de la même manière que pour la méthode absurde.

Algorithme 8 - basé sur le tri par consommation décroissante de ressources
Entrée : liste des activités, liste des ressources consommées par chaque activité
Sortie : liste des activités triée selon la consommation décroissante de ressources

```
compteur = 0  ← on initialise compteur à 0
Pour i dans activites  ← on parcourt la liste des activités qui ont une durée différente de 0
   iMax = activites[i]  ← iMax correspond au numéro de l'activité d'indice i
   compteur = i+1  ← compteur correspondant à l'indice de l'activité i
   Pour j allant de i+1 à longueur activites  ← on parcourt les éléments situés après celui d'indice i
      numero = activites[j]  ← numero correspond au numéro de l'activité d'indice j
      Si ressources[numero-1] > ressources [iMax-1]  ← si l'élément d'indice j consomme plus que
                                                       celui d'indice iMax
         iMax = numero  ← iMax prend la valeur du numéro de l'élément d'indice j
         res = compteur  ← res prend la valeur de l'indice j
      compteur = compteur + 1  ← on passe à l'activité d'indice suivant
   activite_a_echanger = activites[i]  ← numéro de l'activité d'indice i
   Si (ressources[iMax-1] > ressources[activite_a_echanger-1])  ← si l'activité d'indice i consomme
                                                                  moins que l'activité d'indice iMax
         tmp = activites[i]              ← on échange ces deux activités (celles d'indice i et d'indice res)
         activites[i] = activites[res]     en utilisant une variable temporaire tmp
         activites[res] = tmp
```

Dans notre exemple, après avoir appliqué l'algorithme de tri sur la consommation décroissante de ressources, on obtient la liste suivante : activites = [1, 4, 2, 3].
On place alors les activités en suivant l'ordre de cette nouvelle liste.
On place en premier l'activité 1, ce qui nous donne :
ressources_disponibles = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 10, ..., 10].
On place ensuite l'activité 4 du temps t = 11 au temps t = 16 et on obtient
ressources_disponibles = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 4, 4, 4, 4, 10, ..., 10].
On place l'activité 2 du temps t = 17 au temps t = 22 et on a : ressources_disponibles = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 10, ..., 10].
La condition de précédence pour l'activité 3 est vérifiée et on peut la placer du temps t = 17 au t = 22 car il y a assez de ressources disponibles. La liste des ressources disponibles devient donc :
ressources_disponibles = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 4, 4, 4, 4, 0, 0, 0, 0, 0,, 10, ..., 10].
La durée totale de notre projet est ainsi égale à t_{final} = 22. Le résultat retourné par cet algorithme est donc plus intéressant que celui obtenu suite aux deux heuristiques précédentes.

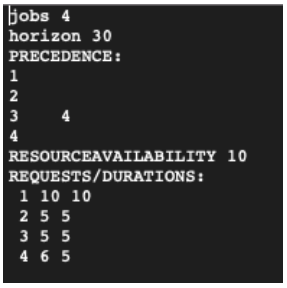


Figure 2 - Rappel du jeu de données de notre exemple

3.5. Recherche locale

Dans cette section, nous verrons un autre type d'algorithme qui prend en compte la solution obtenue par un des algorithmes ci-dessus et qui va la modifier itérativement pour tenter de l'améliorer. Pour ce faire, l'algorithme de la recherche locale va prendre la liste des activités finies dans l'ordre (obtenue grâce à une des heuristiques) et va effectuer toutes les permutations possibles de deux éléments de cette liste. Pour chaque permutation, on va alors d'abord vérifier que cette nouvelle solution est valable (respect des précédences), puis on va calculer la nouvelle durée associée. Parmi toutes les nouvelles solutions valables trouvées, on prend celle qui donne la plus petite durée et on compare cette durée à celle obtenue par la solution de départ. Si la nouvelle durée est moindre que celle de départ, alors cette nouvelle solution devient celle de référence et on recommence un nouveau cycle. L'algorithme continuera de tourner jusqu'à ce qu'aucune modification n'améliore le résultat. Prenons un exemple.

Supposons qu'une des heuristiques nous donne le résultat suivant:
liste des activités = [1 , 2 , 3 , 4 , 5 , 6]
durée totale = 20

Nous allons donc effectuer toutes les permutations possibles, puis, après avoir vérifié que la solution est toujours valable, calculer de nouveau la durée totale. Notons que dans tous les jeux de données qui nous ont été donnés, la première et dernière activités restent toujours fixes. Il est donc inutile de réaliser les permutations pour ces activités là (la solution serait automatiquement non valable).

Permutation	Nouvelle liste avec permutation	Nouvelle durée totale
2 - 3	[1 , 3 , 2 , 4 , 5 , 6]	25
2 - 4	[1 , 4 , 3 , 2 , 5 , 6]	Solution non valable car le respect des précédences n'est plus respecté
2 - 5	[1 , 5 , 3 , 4 , 2 , 6]	18
3 - 4	[1 , 2 , 4 , 3 , 5 , 6]	22
3 - 5	[1 , 2 , 5 , 4 , 3 , 6]	24
4 -5	[1 , 2 , 3 , 5 , 4 , 6]	Solution non valable car le respect des précédences n'est plus respecté

Nous observons qu'en réalisant la permutation des activités 2 et 5, nous obtenons une meilleure durée qu'avec la solution de départ. Nous allons donc prendre cette nouvelle solution et refaire toutes les permutations. Ce processus sera répété jusqu'à ce que les permutations réalisées sur une solution de départ n'améliorent pas la durée de cette solution.

Voyons maintenant comment fonctionne l'algorithme, étape par étape.

Etape 1 - On prend la solution de référence et on effectue les permutations

---

Algorithme 9
Entrée : liste de référence des activités finies dans l'ordre
Sortie : liste de référence permutée

```
solutions_testees = [solution_de_départ]  ← on crée une nouvelle liste qui contiendra toutes les solutions
                                            testées jusqu'à présent. On l'initialise avec la solution de départ.
                                            Cette instruction ne se fait qu'une seule fois, au tout début du programme.
Pour i allant de 1 au nombre d'activités-1  ← on commence à 1 et on finit au nombre d'activités -1
                                              car l'activité de départ et d'arrivée sont fixes
    Pour j allant de 1 au nombre d'activités-1
        Si i==j  ← si les indices i et j sont les mêmes
            continue  ← on passe au prochain indice j car on obtiendrait la même liste donc inutile
        Si durée de l'activité en i = 0 ou durée de l'activité en j = 0 ← si l'une des activités a une durée nulle,
            continue                                                      il est inutile d'effectuer la permutation car il n'y
                                                                          aura pas de changement
        solution_a_tester = solution_référence[:] ← on crée une nouvelle variable dans laquelle on mettra la
                                                    nouvelle solution permutée
        tmp = solution_a_tester[i]
        solution_a_tester[i] = solution_a_tester[j]
        solution_a_tester[j] = tmp  ← à la fin de ces 3 lignes, les activités d'indice i et j ont été permutées
        Si solution_a_tester est dans solutions_testees ← si la solution a déjà été testée, inutile de continuer
            state = "break"
        Si state=="break"
            continue  ← on passe au prochain indice j
        solutions_testees = solutions_testees + solution_a_tester ← sinon, on ajoute la solution en cours à la
                                                                    liste des solutions déjà testées
```

Etape 2 - On vérifie si les précédences sont toujours respectées

Pour cette étape nous n'allons pas détailler l'algorithme car il ressemble fortement à l'algorithme 2 décrit précédemment.
Il est juste important de noter que si, dans cette nouvelle solution, une des activités n'a pas pas tous ses prédécesseurs placés avant elle, alors le programme ne continue pas et passe directement au prochain indice j, car la solution n'est pas valable.

Etape 3 - On place toutes les activités le plus tôt possible

De même, cet algorithme ressemble fortement à l'algorithme 3, donc nous n'allons pas le détailler ici.
Maintenant que nous savons que cette solution est valable (étape 2) toutes les activités sont placées le plus tôt possible.
En sortie, nous obtenons le temps de début de chaque activité.

Etape 4 - Calcul de la nouvelle durée

Algorithme 10
Entrée : liste des débuts de chaque activité, liste des durées de chaque activité
Sortie : durée totale

```
durée = 0
Pour  i allant de 0 au nombre d'activités ← on commence à 1 et on finit au nombre d'activités -1
                                            car l'activité de départ et d'arrivée sont fixes
    Si (fin_activité[i]) > durée:
        durée = fin_activité[i]
retourner durée ← on a ainsi trouvé la plus grande date de fin pour parmi toutes les activités, ce qui correspond à
                  la durée totale
```

Etape 5 - Détermination de la meilleure solution du cycle en cours et fin

Tout au long du cycle, lorsqu'une solution valable est trouvée, on la compare avec la précédente meilleure solution du cycle (qui est stockée dans une variable). Si la nouvelle solution est meilleure, alors celle-ci devient la "meilleure" solution du cycle et on stocke dans des variables la durée, les dates de début et de fin des activités et les activités triées dans l'ordre.

Une fois que le cycle est fini, il y aura une autre comparaison, entre la meilleure durée du cycle et la durée de la solution de référence. Si la durée de la solution de référence est la même que la nouvelle durée trouvée, alors le programme s'arrête et retourne la solution de référence. Sinon, c'est cette nouvelle solution qui devient la solution de référence et le programme redémarre à l'étape 1.

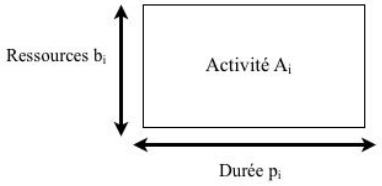
Cet algorithme a été placé dans les six programmes des différentes heuristiques, afin de pouvoir améliorer les résultats, si c'est possible.

4. Calcul du minorant

Dans cette section, le but est de trouver un minorant optimal, c'est à dire le plus grand possible. Ce minorant nous permettra ensuite d'évaluer les solutions obtenues par les différents heuristiques: plus les résultats trouvés sont éloignés du minorant, moins ils sont bons. Nous présenterons pour cela trois minorants. Le premier minorant est géométrique, le second se trouve en faisant des chemins de précédence, et le troisième se trouve en cherchant à faire des empilements. Nous verrons que selon les données, ça ne sera pas toujours le même minorant qui sera le meilleur.

4.1. Minorant géométrique

Le minorant géométrique se trouve en représentant chaque activité sous forme d'une aire.



Il suffit donc d'additionner les aires de toutes les activités pour trouver l'aire totale de toutes les activités. Puis pour trouver la durée, on divise l'aire totale par la ressource maximale. On note b_i la ressource utilisée par l'activité i, p_i la durée de l'activité i, b_{max} la ressource maximale et n le nombre d'activités. Le calcul effectué par l'algorithme est le suivant.

$$\frac{\sum_{i=0}^{n} b_i.p_i}{b_{max}}$$

Notons que, dans ce problème, la durée totale est toujours un nombre entier. Donc pour être plus précis, on regarde si le résultat obtenu est entier. Si c'est le cas, alors c'est le minorant, et si ce n'est pas le cas, on prend la partie entière du résultat et on y ajoute 1.

Voici ci-dessous l'algorithme qui permet de trouver le minorant géométrique.

Algorithme 11
Entrée : liste de la durée des activités, liste des ressources des activités, ressource max
Sortie : La valeur du minorant géométrique

```
aire_totale = 0    ← on initialise l'aire à 0
Pour i allant de 0 au nombre d'activités
        aire_totale = aire_totale + durée[i] * ressources[i]   ← si il y a une condition de précédence à respecter
Si aire_totale // ressource_max != aire_totale / ressource_max      ← si le résultat n'est pas un entier
        minorant_geom = aire_totale // ressource_max + 1  ← on prend la partie entière et on ajoute 1
Sinon     ← le résultat est un entier
        minorant_geom = aire_totale // ressource_max    ← on n'a pas besoin de modifier la valeur
```

4.2. Minorant par chemin de précédences

Il est également possible de représenter les données du problème sous forme d'un graphe, où chaque activité (représentée par un sommet) est reliée à ses précédents. Pour trouver le minorant, il va falloir trouver tous les chemins de précédence possibles, additionner les durées des activités de ce chemin et garder le chemin qui donne le plus grand résultat.

Prenons un exemple. Sur le graphe ci-dessous, les chiffres noirs représentent les activités, les chiffres rouges représentent la durée de l'activité au-dessus de laquelle ils se trouvent et les flèches représentent les précédences (par exemple : les prédécesseurs de l'activité 3 sont 2 et 5).

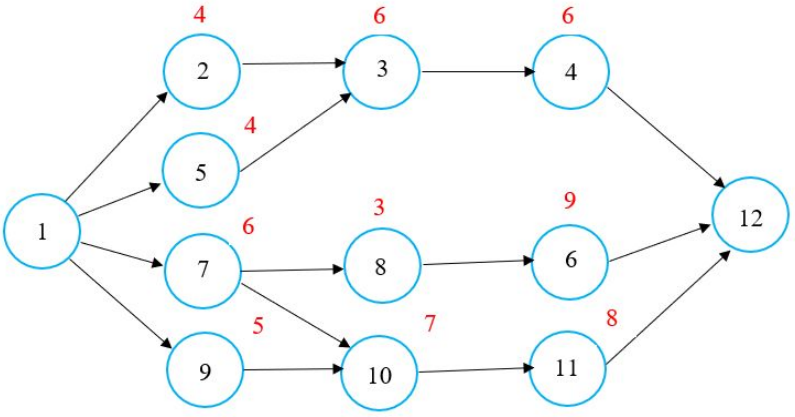


Figure 6: Exemple - Données représentées sous forme d'un graphe

Sur ce graphe, on peut facilement trouver le minorant. Il y a 5 chemins de précédence différents, et on voit rapidement que le chemin qui nous donne la plus grande durée totale est (7,10,11) (avec une durée de 21). Cependant, avec un grand nombre de données, il n'est plus possible de trouver ce minorant à la main.

On a donc écrit un algorithme qui change la valeur des durées de chaque activité en son temps de fin minimal. Pour ce faire, on prend chaque activité, on vérifie que tous ses précédents ont déjà leur durée définitive, et si c'est le cas, la nouvelle valeur de la durée de cette activité est :ancienne durée + la plus grande durée parmi celles de ses précédents. Sinon, on passe à la prochaine activité. On répète ceci jusqu'à ce que toutes les activités aient été traitées. Enfin, en prenant l'activité qui a la plus grande durée, on trouve le minorant.

Algorithme 12

Entrée : liste de la durée des activités, liste de précédence des activités
Sortie : La valeur du minorant par chemin

```
activites_modifiees = []   ← on crée une liste dans laquelle on met toutes les activités dont on a modifié
                             la durée
durees_modifiees = duree[:] ← on crée une liste qui contiendra les durées modifiées de toutes les activités
Tant que longueur(activites_modifiees) != longueur(duree)  ← tant que toutes les activités ne sont pas
                                                             dans la liste activites_modifiees
        Pour i allant de 0 à longueur(duree)
            Si duree[i]==0   ← si l'activité i+1 a une durée nulle, on ne change pas sa valeur car sinon,
                               le résultat serait faussé
                durees_modifiees[i] = 0
                activites_modifiees += [i+1]
                continue     ← on passe directement à l'activité suivante
        state = ''activite non modifiee''   ← state nous dit si l'activité en cours a déjà été traitée ou non
            Pour k dans activites_modifiees
                Si k == i + 1      ← si l'activité modifiée k est la même que l'activité i+1
                state = ''activite modifiee''          ← alors l'activité i+1 a déjà été traitée
        Si state == ''activite modifiee''
                continue                ← donc on arrête cette boucle et on passe au prochain i
        maxi = 0
        a = []  ← liste dans laquelle on mettra les prédécesseurs de l'activité (i+1) qui ont déjà été modifiés
            Pour p dans precedence[i]:
                Pour k dans activites_modifiees:
                    Si p == k:  ← si le prédécesseur p de l'activité (i+1) est dans la liste des activités modifiées
                            a += [p]  ← alors on met p dans la liste a
            Si a == precedence[i]: ← si tous les prédécesseurs de l'activité (i+1) ont été modifiés
                Pour j in precedence[i]:
                    Si durees_modifiees[j-1] > maxi :
                            maxi = durees_modifiees[j-1]  ← alors la variable maxi prend la plus grande durée des
                                                            prédécesseurs de l'activité (i+1)
                durees_modifiees[i] += maxi  ← on change la valeur de la durée de l'activité (i+1)
                activites_modifiees += [i+1]  ← on ajoute l'activité (i+1) à la liste des activités modifiées
minorant = 0   ← on initialise la valeur du minorant
Pour i allant de 0 à longueur(duree)
        Si durees_modifiees[i] > minorant:
            minorant = durees_modifiees[i]  ← minorant prend la plus grande durée parmi la liste des durées
```

4.3. Minorant par empilements

Ce minorant se trouve en faisant une représentation géométrique du problème, qui est un peu différente du premier minorant trouvé. Nous allons en fait placer les unes après les autres toutes les activités qui ne peuvent pas s'empiler. Pour se faire, l'algorithme va d'abord ordonner les activités dans l'ordre décroissant des durées, puis des ressources. Si l'activité maintenant en première position (qui demande le plus de ressources) a besoin de plus de la moitié de la ressource maximum disponible comme ressource, on la place. Puis on passe à la deuxième activité. Si cette activité peut être placée au dessus de l'activité précédente, on arrête l'algorithme, sinon on la place juste après la première. On continue ainsi jusqu'à ce qu'une activité puisse être placée au dessus de la précédente. On additionne ensuite les durées de toutes les activités qui n'ont pas pu être empilées.

On pourrait s'arrêter ici, mais nous avons trouvé une façon de légèrement améliorer ce minorant. Une fois que toutes les activités non empilables ont été placées, on regarde si l'activité suivante a le même besoin en ressources que la dernière placée. Si c'est le cas, et si ce besoin en ressources ne peut pas être empilé plus de 2 fois sur lui-même, alors on additionne les durées de toutes les activités qui ont ce besoin en ressource (y compris la durée de l'activité qui avait déjà été placée). On divise alors la somme obtenue par 2, puisqu'on peut placer 2 activités l'une sur l'autre en chaque temps. Si ce résultat est plus grand que la durée de la dernière activité placée, alors on enlève au minorant la durée de cette activité et on ajoute ce résultat. Cette partie de l'algorithme sert par exemple dans un cas où ressource_max = 14, et où il y a des activités avec des besoins de ressources de 8 et de 7 (puisque 8+7>14 et 7+7<=14).

Voyons maintenant à quoi ressemble l'algorithme. On enlève d'abord les activités qui ont une durée nulle. On effectue ensuite deux tris à bulle pour obtenir toutes les durées, puis les ressources dans l'ordre décroissant. On modifie également la liste des durées pour que les indices des ressources et des durées correspondent. On ne détaillera pas cette partie de l'algorithme car elle reprend simplement un algorithme vu en cours.

Algorithme 13

Entrée : liste de la durée des activités triées, liste des ressources des activités triée, ressource max
Sortie : La valeur du minorant par empilement

```
state = "continuer"          ← l'état nous dit si on doit continuer l'algorithme
minorant_empilement = 0  ← on crée une valeur contenant le minorant
Si ressources_triees[0] > ressource_max / 2:   ← si la valeur des ressources utilisées par l'activité qui en utilise le
                                                plus est supérieure à la ressource max divisée par 2, on peut commencer l'algorithme
    minorant_empilement = durees_triees[0]  ← on initialise la valeur du minorant avec la durée de l'activité qui
                                              a besoin du plus de ressource
    Tant que  state != "fini":
        Pour i allant de 1 au nombre d'activités
            Si ressources_triees[o] > (ressource_max - ressources_triees[o-1])  ← si il n'y a pas assez de
                                                                                  place pour placer l'activité i au dessus de la précédence
                minorant_empilement += durees_triees[i] ← on ajoute au minorant la valeur de la durée de
            Sinon si  ressources_triees[o]==ressources_triees[o-1] et (ressource_max -
            2*ressources_triees[o]) < ressources_triees[o] ←Si l'activite o a la même ressource que l'activité
                                                            d'avant et qu'on ne peut pas empiler plus de deux fois l'activité o sur elle-même,
                p=o ← nouvelle variable qui contiendra l'indice de l'activité qu'on est en train de tester
                somme = durees_triees[o-1] ← variable qui contiendra la somme de toutes les durées des
```

```
                                    activités qui ont le même besoin en ressource que l'activité o (dont o-1)
        Tant que  ressources_triees[p]==ressources_triees[o]← On fait tourner le programme tant
                                        que l'activité p a le meme besoin en ressource que celle de départ o
                somme += durees_triees[p]
                 p+=1
         somme = int((somme-0,5)/2+1 ← On divise la somme par 2 (puisqu'on peut placer 2
                activités l'une sur l'autre en chaque temps) tout en s'assurant que le résultat est entier
         Si durees_triees[o-1]<somme←si le résultat obtenu est plus grand que la durée de l'activité
                                                                                                    o-1
                minorant_empilement += (somme-durees_triees[o-1])←On enlève au minorant la
    durée de l'activité o-1 et on rajoute la durée minimale des activités avec les mêmes ressources

         state = "fini"←On arrête le programme
         break
    Sinon
         state = “fini”   ← sinon on arrête de faire tourner l’algorithme
         break
```

En fonction des données, ce n’est pas toujours le même minorant qui est le meilleur. En effet, sur 37 jeux de données, le minorant géométrique est plus grand dans 19 cas, le minorant par chemin de précédences dans 12 cas, le minorant par empilements dans 4 cas et les minorants géométrique et par chemin ont la même valeur dans 2 cas. Il est donc important de toujours calculer les trois minorants pour obtenir la meilleure valeur possible.

En général, dans un jeu de données où il y a un grand nombre d’activités qui ont une durée nulle, le minorant géométrique est plus intéressant. En revanche, dans un jeu de données où toutes les activités ont beaucoup de précédences et des petites consommations de ressources, c’est le minorant par chemin qui est le plus intéressant. Enfin, dans un jeu de données il y a un grand nombre d’activités qui ont des besoins en ressources élevés, c’est en général le minorant par empilements qui donne le meilleur résultat.

Pour nos résultats, nous avons donc, pour chaque jeu de données, pris le plus grand minorant parmi les trois.

5. Expérimentations et résultats

Nous avons implémenté nos algorithmes dans le langage Python. Nous avons testé les algorithmes présentés à la section 3 sur le jeu de données fourni pour le projet. Ce jeu de données comporte 37 cas test ayant soit 32 tâches à ordonnancer, soit 122. La borne maximale B_{max} de ressources utilisables est comprise entre 9 et 43. L’ensemble des algorithmes de résolution ont un temps de calcul équivalent. Ce critère n’étant pas pertinent, nous allons baser l’évaluation des heuristiques sur d’autres critères.

Le premier but de ces expérimentations est de comparer les solutions retournées par les différents algorithmes de résolution, tout en cherchant à savoir lequel d’entre eux traite le problème de la manière la plus optimale. L’algorithme qui retourne la solution la plus proche de la valeur du minorant calculé dans la section 4 est donc considéré comme le plus satisfaisant.

Avant de comparer les résultats obtenus, nous avons regroupé dans un tableau (annexe 7.2 et fichier Résultats.csv) le minorant ainsi que l’ensemble des dates de fin du projet pour chaque algorithme.

5.1. Première partie des expérimentations : Comparaison des Heuristiques 1 à 6

Dans cette première partie, nous allons comparer les résultats retournés par les six premiers algorithmes, c'est-à-dire ceux relevant des méthodes absurde, de séparation de liste et de tri de liste. Nous allons dans un premier temps comparer de manière générale l'ensemble des solutions retournées par chaque algorithme. Puis dans un second temps, nous comparerons les valeurs de chacune des solutions obtenues par chaque heuristique, et ce pour chaque donnée du jeu mis à notre disposition.

Dans un premier temps, nous avons calculé la moyenne et l'écart-type du minorant et des dates de fin du projet pour chaque heuristique, ce qui nous donne le tableau suivant. Nous pouvons ainsi comparer de manière générale les valeurs retournées par les différents algorithmes avec le minorant.

	Minorant	H_1	H_2	H_3	H_4	H_5	H_6
Moyenne	67,97	78,84	78,84	79,95	79,92	79,19	80,51
Ecart-type	47,95	55,95	55,95	57,96	57,74	55,95	56,95

H_ 1 - Heuristique idiote
H_2 - Heuristique de séparation de listes
H_3 - Heuristique de tri par consommation décroissante de ressources
H_4 - Heuristique de tri par consommation croissante de ressources
H_ 5 - Heuristique de tri par durée décroissante
H_6 - Heuristique de tri par durée croissante

Comme nous l'avions remarqué dans notre exemple de la section 4, les algorithmes 1 et 2 retournent les mêmes résultats pour l'ensemble des données. Il s'agit d'ailleurs des heuristiques qui renvoient les meilleurs résultats d'un point de vue général puisque la moyenne des dates de fin du projet de ces algorithmes est celle qui est la plus proche de la moyenne du minorant. Elles possèdent d'ailleurs l'écart-type le plus faible, qui est égal à celui de l'heuristique 5.
Les 4 heuristiques basées sur le tri de liste renvoient de moins bons résultats dans l'ensemble par rapport aux deux premières. La moyenne et l'écart type des algorithmes 3 et 4 sont quasi similaires et un peu plus élevés que les valeurs obtenues aux méthodes 1 et 2. L'heuristique 5 possède la moyenne la plus basse de tous les algorithmes de tri, alors que la moyenne de l'heuristique 6 est la plus élevée. L'écart-type de ces deux algorithmes est cependant plus faible que celui des heuristiques 3 et 4. Les algorithmes de tri selon la durée retourneraient donc des solutions moins éparpillées que les algorithmes basés sur un tri en fonction des ressources.
On peut donc penser, suite à cette première observation, que l'heuristique 5 est l'algorithme le plus intéressant parmi tous les algorithmes de tri, car il possède la moyenne et l'écart-type les plus bas. Nous pouvons noter tout de même que la différence entre tous ces algorithmes est assez faible : la moyenne varie seulement de 78,84 à 80,51.
D'ailleurs, pour les deux paramètres (consommation de ressources et durée), le tri décroissant de la liste des activités retourne des résultats plus intéressants que le tri croissant. Ce résultat est expliqué par le fait qu'il est plus difficile de placer tôt une activité dont la consommation de ressources est élevée ou qui a une durée importante. En effet, en plaçant en premier les activités qui consomment le plus ou qui durent le plus longtemps, on aura plus de chance de pouvoir réaliser plusieurs activités en même temps. Ainsi, moins de ressources seront perdues et le projet pourra se terminer plus vite, minimisant ainsi la date de fin du projet. Cependant, la contrainte de précédence pose problème dans ces 4 algorithmes de tri car on ne pourra pas vraiment placer les activités dans l'ordre initial de la liste triée.

Maintenant, comparons les valeurs obtenues par chaque heuristique pour chacune des données. Tout d'abord, regardons le nombre de fois où chaque méthode est la meilleure parmi celles testées.

Algorithmes retournant la meilleure solution	Nombre de données	Pourcentage
Heuristique 1 et 2	6/37	16,2 %
Heuristique 3	5/37	13,5 %
Heuristique 4	2/37	5,4 %
Heuristique 5	5/37	13,5 %
Heuristique 6	2/37	5,4 %
Toutes les algorithmes retournent la même solution	8/37	21,6 %
Plusieurs algorithmes de la méthode de tri renvoient la même solution, qui est meilleure que celle retournée par les heuristiques 1 et 2	6/37	16,2 %
Plusieurs algorithmes retournent la même solution	3/37	8,1 %

Tout d'abord, on remarque que les heuristiques 4 et 6 renvoient moins souvent un meilleur résultat que les heuristiques 3 et 5. Le tri décroissant semble donc être le plus probant.
De plus, dans 45,9 % des cas, plusieurs algorithmes retournent la meilleure solution pour une donnée, ce qui explique pourquoi les moyennes et les écarts-types calculés plus tôt sont assez proches.
Cette observation ne nous permet pas de conclure sur la meilleure heuristique, mais elle confirme tout de même les résultats précédents : les algorithmes 1, 2 et 5 sont les plus intéressants pour la résolution de notre problème.

Calculons maintenant pour chaque donnée l'écart relatif vis à vis de la meilleure solution ainsi que l'écart relatif vis à vis du minorant. Le premier, appelé également gap, se calcule de la manière suivante.
Soient z la valeur de la solution retournée par l'algorithme et z^{best} la valeur de la meilleure solution obtenue, alors gap = $\frac{z - zbest}{z}$.

Le second, appelé gap maximum, est déterminé à partir de la formule suivante.
Soient z la valeur de la solution retournée par l'algorithme et min la valeur du minorant, alors
$gap_{max} = \frac{z - min}{z}$
Voici un tableau récapitulant l'ensemble des résultats obtenus pour chaque heuristique. Les valeurs correspondent à la moyenne et à l'écart-type du gap et du gap maximum pour chacune des données.

	Gap		Gap maximum	
	Moyenne	Ecart-type	Moyenne	Ecart-type
Heuristique 1	3%	5%	13%	8%
Heuristique 2	3%	5%	13%	8%
Heuristique 3	3%	4%	12%	8%
Heuristique 4	4%	4%	13%	9%
Heuristique 5	3%	5%	12%	9%
Heuristique 6	5%	6%	14%	9%

Ce tableau confirme les conclusions tirées des données précédentes. On peut déjà confirmer le fait que les heuristiques 4 et 6 sont moins intéressantes que les autres car la moyenne de leur gap et de leur gap maximum est plus importante que pour le reste des algorithmes. Les solutions retournées par l'algorithme 6 sont d'ailleurs en moyenne plus dispersées, Nous pouvons donc mettre de côté ces deux méthodes car elles renvoient des résultats de moins bonne qualité que les autres algorithmes.
Le calcul de l'écart relatif vis-à-vis du minorant nous donne de nouvelles informations sur les algorithmes 1, 2 et 5, que nous avions considéré plus tôt comme les plus performants. En effet, on remarque que l'algorithme 5 renvoie plus souvent une solution proche du minorant que les deux premières méthodes (gap maximum de 12% en moyenne contre 13%). Cependant, les solutions retournées par cet algorithme sont plus dispersées (écart-type du gap maximum à 9% en moyenne contre 8%), ce qui explique pourquoi la moyenne des solutions retournées par les algorithmes 1 et 2 est plus basse que celle de la méthode 5.
Par ailleurs, on peut constater avec ces résultats que l'heuristique 3 est la meilleure méthode puisque la valeur moyenne de son gap est la même que celle de l'heuristique 5 (3%) et que les solutions retournées sont moins dispersées (écart-type du gap de 4% contre 5%). La moyenne et l'écart-type de son gap maximum sont également les plus basses de toutes les heuristiques. L'algorithme 3 renvoie donc plus souvent des valeurs proches du minorant.

L'ensemble des observations effectuées met en avant le fait que les algorithmes retournent des solutions assez proches. Cependant, les algorithmes de tri par consommation décroissante de ressources (algorithme 3) et par durée décroissante (algorithme 5) sont les deux méthodes de résolution les plus intéressantes. En effet, même si les algorithmes 1 et 2 nous donnent en général les meilleurs résultats, ils renvoient souvent des valeurs assez dispersées dont certaines qui sont assez loins du minorant. Les heuristiques 3 et 5, quant à elles, reposent sur un tri préalable de la liste des activités et permettent de placer de manière plus réfléchie les activités, que ce soit en fonction de leur consommation de ressources ou de leur durée. La méthode 3 retourne moins souvent les meilleures solutions, mais il s'agit de l'algorithme le plus constant car l'écart-type de son gap est le plus faible. Les valeurs retournées sont donc moins dispersées. L'algorithme 5, quant à lui, renvoie à lui seul la meilleure solution dans 13,5% des cas. La moyenne de l'ensemble des valeurs retournées par cet algorithme est la plus basse de tous les algorithmes de tri. Cependant, les solutions sont un peu plus dispersées que pour l'algorithme 3.
On peut donc conclure, suite à ces résultats, que les méthodes 3 et 5 sont les pertinentes pour résoudre ce problème car elles renvoient en moyenne des résultats intéressants, qui de plus ne sont pas trop dispersés.
Les algorithmes 4 et 6 sont à mettre de côté car les résultats fournis par ces deux heuristiques sont de moins bonne qualité, que ce soit en terme de valeur retournée ou bien de dispersion des résultats.

5.2. Deuxième partie des expérimentations : Comparaison des résultats obtenus avec ceux issus de la recherche locale

Afin d'améliorer les résultats obtenus par ces algorithmes, nous avons ensuite créé un algorithme de recherche locale. Analysons maintenant les résultats suite à cet ajout.
Nous avions 37 jeux de données à résoudre avec 6 heuristiques différentes. Sur les 222 lancements des programmes, la recherche locale a permis d'améliorer le résultat dans 149 des cas, soit 67%. Nous pouvons maintenant comparer de manière générale les valeurs retournées par les différents algorithmes complétés d'une recherche locale avec le minorant.

	Minorant	H_1 avec recherche locale	H_2 avec recherche locale	H_3 avec recherche locale	H_4 avec recherche locale	H_5 avec recherche locale	H_6 avec recherche locale
Moyenne	67,97	76,00	76,00	76,62	76,76	76,76	77,08
Ecart-type	47,95	55,21	55,21	56,58	56,31	55,60	55,76

H_ 1 - Heuristique idiote
H_2 - Heuristique de séparation de listes
H_3 - Heuristique de tri par consommation décroissante de ressources
H_4 - Heuristique de tri par consommation croissante de ressources
H_ 5 - Heuristique de tri par durée décroissante
H_6 - Heuristique de tri par durée croissante

On voit que par rapport au tableau obtenu sans recherche locale, il y a une nette baisse des moyennes. Elles varient de 78,84 à 80,51 sans la recherche locale et varient de 76,00 à 77,08 avec l'ajout de la recherche locale. Les moyennes ont toutes diminuées d'environ 3. On observe également une amélioration au niveau des écart-types, d'environ 1 pour chaque heuristique. Dans l'ensemble, les observations faites grâce aux heuristiques sans recherche locale ne changent pas : les heuristiques 1 et 2 nous donnent la meilleure moyenne, l'heuristique 6 semble la moins intéressante et l'heuristique 3 semble la plus intéressante parmi les heuristiques de tri.

Nous pouvons également voir dans le tableau ci-dessous que les gaps maximum sont nettement diminués suite à cet ajout.

	Gap maximum	
	Moyenne	Ecart-type
Heuristique 1 avec RL	9%	7%
Heuristique 2 avec RL	9%	7%
Heuristique 3 avec RL	9%	7%
Heuristique 4 avec RL	9%	8%
Heuristique 5 avec RL	9%	8%
Heuristique 6 avec RL	10%	7%

RL - Recherche locale

Dans l'ensemble, les moyennes du gap maximum diminuent de 3 à 4%. Pour les écarts-types, on voit simplement une légère amélioration avec une baisse de 1 à 2% pour chaque heuristique.

Voyons maintenant comment la recherche locale permet d'améliorer les résultats dans l'ensemble. Sur les 37 jeux de données, la recherche locale nous permet d'améliorer le meilleur résultat dans 20 des cas, soit 54%. Le tableau ci-dessous nous permet de comparer plus en détails les meilleures solutions obtenues sans et avec recherche locale.

	Résultats		Gap maximum	
	Moyenne	Ecart-type	Moyenne	Ecart-type
Meilleures solutions sans RL	76,70	55,21	9%	7%
Meilleures solutions avec RL	74,78	54,36	7%	6%

RL - Recherche locale

Nous pouvons donc voir que l'ajout de la recherche locale dans les 6 heuristiques nous permet de diminuer la moyenne des meilleurs résultats de 1,92, ce qui nous permet de nous approcher de la moyenne du minorant (qui est de 67,97).
De plus, sur les 37 jeux de données, nos algorithmes permettent de trouver une solution exactement égale au minorant dans 12 cas (32%), ce qui est très satisfaisant.
Cet algorithme paraît donc très intéressant, puisqu'il nous permet de nettement améliorer nos résultats.
Cependant, un point négatif de cet algorithme est que, dans certains cas, il met beaucoup de temps (jusqu'à 5 mins).

6. Conclusion

Pour réaliser ce projet, nous avons créé six heuristiques différentes qui nous permettent de trouver des solutions au problème de RCPSP. Suite à cela, nous avons écrit un algorithme de recherche locale, qui tente d'améliorer les résultats obtenus en testant des permutations de position des activités. Nous avons également calculé trois minorants, un géométrique, un autre qui se base sur des chemins de précédence et un dernier qui se trouve en additionnant les durées de toutes les activités qui ne peuvent pas s'empiler.

Suite à nos expérimentations, nous avons pu constater que même si les heuristiques 1 et 2 nous renvoient en moyenne les meilleurs résultats, ce sont les heuristiques 3 et 5 qui sont les plus pertinentes car elles renvoient en moyenne des résultats intéressants, qui de plus ne sont pas trop dispersés. Les heuristiques 4 et 6 quant à elles, peuvent être mises de côté car les résultats fournis sont de moins bonne qualité. De plus, la recherche locale est très intéressante, car elle nous permet d'obtenir de meilleurs résultats pour toutes les heuristiques, et donc de nous rapprocher le plus possible du minorant.

Dans l'ensemble, les résultats paraissent assez satisfaisant. En effet, lorsque nous prenons les meilleurs résultats obtenus pour chaque jeu de données, la moyenne des gaps maximum est de 7% seulement, et avec un écart type de 6%, ce qui paraît très faible. Il serait tout de même possible d'imaginer qu'avec un meilleur minorant, ce gap serait encore plus faible. La recherche d'un autre minorant pourrait donc être une suite possible à notre projet.

Ce projet nous a permis de fortement améliorer nos compétences en informatique, qui étaient jusqu'à maintenant, très basiques. De plus, la forme assez ouverte de ce projet nous a non seulement fait gagner en initiative et en autonomie mais il nous a également fallu beaucoup de créativité pour trouver les différentes heuristiques de résolution ou les minorants par exemple.

7. Annexes

7.1. Lecteur de données

ENTRÉE	SORTIE
"Fichier.txt" jobs 4 horizon 30 PRECEDENCE: 1 2 3 4 4 RESOURCEAVAILABILITY 10 REQUESTS/DURATIONS: 1 10 10 2 5 5 3 5 5 4 6 5	L'algorithme retourne plusieurs listes : - Une liste regroupant le numéro de chaque activité à ordonnancer : liste activites = [1,2,3,4] - Une liste contenant l'ensemble des durées associées à chaque activité : liste duree = [10,5,5,5] - Une liste de la quantité de ressources consommées par chaque activité : liste ressources = [10,5,5,6] - Une liste contenant l'ensemble des listes de précédence associées à chaque activité : precedence = [[ ],[ ],[4],[ ]] L'algorithme retourne également certaines données du problème, telles que : - Le nombre d'activités à placer : nb_activites = 4 - La durée maximale Cmax à ne pas dépasser : duree_max = 30 - La quantité de ressources maximale Bmax à utiliser : ressource_max = 10

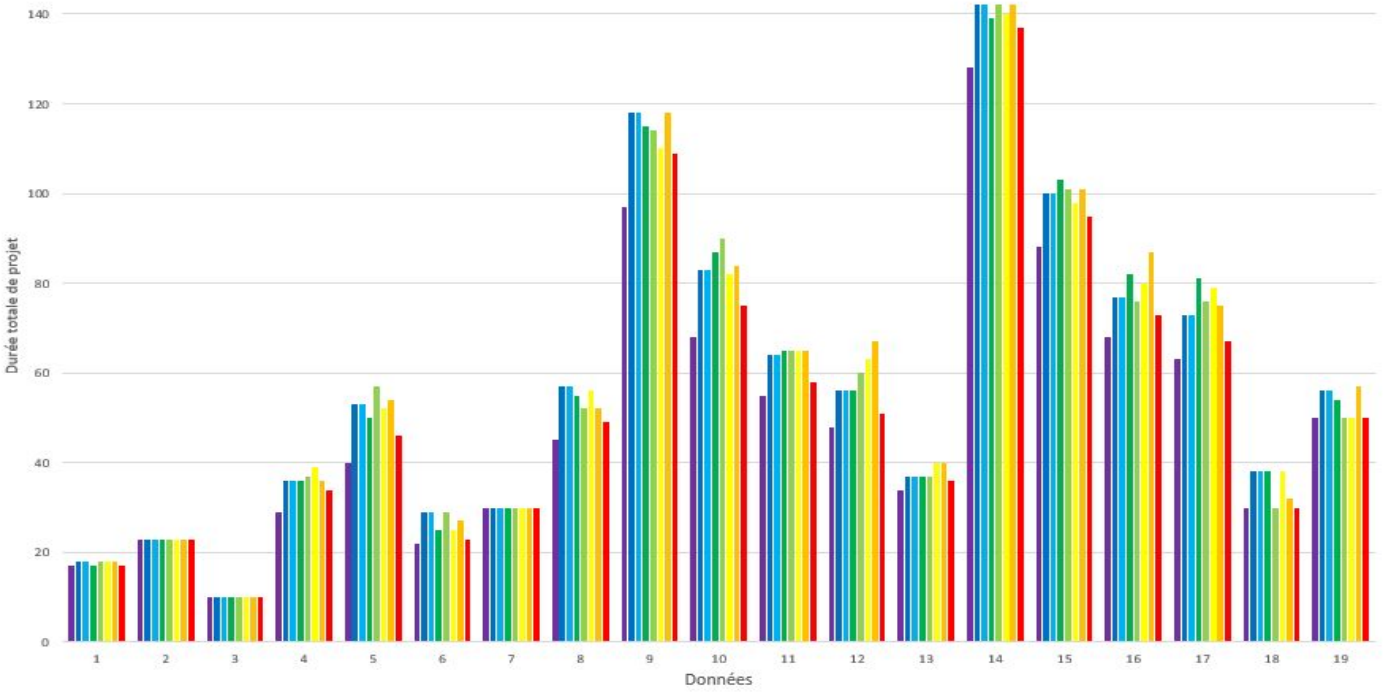
7.2. Résultats détaillés des heuristiques et minorants

Données	Minorant Géom	Minorant Chemin	Minorant Empil	H_1 (RL)	H_2 (RL)	H_3 (RL)	H_4 (RL)	H_5 (RL)	H_6 (RL)
1	15	8	17	18 (17)	18 (17)	17 (-)	18 (17)	18 (17)	18 (17)
2	17	14	23	23 (-)	23 (-)	23 (-)	23 (-)	23 (-)	23 (-)
3	8	10	7	10 (-)	10 (-)	10 (-)	10 (-)	10 (-)	10 (-)
4	29	20	21	36 (35)	36 (35)	36 (34)	37 (35)	39 (35)	36 (-)
5	40	24	23	53 (46)	53 (46)	50 (46)	57 (46)	52 (47)	54 (47)
6	22	19	19	29 (28)	29 (28)	25 (23)	29 (25)	25 (23)	27 (25)
7	16	30	0	30 (-)	30 (-)	30 (-)	30 (-)	30 (-)	30 (-)
8	45	30	8	57 (51)	57 (51)	55 (49)	52 (50)	56 (52)	52 (50)
9	97	21	91	118 (110)	118 (110)	115 (110)	114 (109)	110 (-)	118 (109)
10	68	27	42	83 (76)	83 (76)	87 (76)	90 (75)	82 (78)	84 (78)
11	55	24	0	64 (58)	64 (58)	65 (59)	65 (58)	65 (59)	65 (59)
12	45	19	48	56 (51)	56 (51)	56 (51)	60 (51)	63 (51)	67 (51)
13	34	17	0	37 (-)	37 (-)	37 (36)	37 (-)	40 (36)	40 (36)
14	128	47	98	142 (137)	142 (137)	139 (-)	142 (139)	140 (139)	142 (139)
15	88	28	26	100 (96)	100 (96)	103 (96)	101 (97)	98 (95)	101 (99)
16	68	48	0	77 (76)	77 (76)	82 (76)	76 (73)	80 (-)	87 (78)
17	63	38	0	73 (67)	73 (67)	81 (70)	76 (68)	79 (70)	75 (71)
18	25	30	0	38 (30)	38 (30)	38 (30)	30 (-)	38 (32)	32 (31)
19	31	50	0	56 (51)	56 (51)	54 (52)	50 (-)	50 (-)	57 (55)
20	21	28	0	34 (31)	34 (31)	28 (-)	34 (31)	28 (-)	39 (28)
21	46	51	13	72 (70)	72 (70)	68 (65)	71 (69)	74 (68)	71 (67)
22	37	67	0	70 (68)	70 (68)	71 (68)	67 (-)	67 (-)	71 (69)
23	25	50	0	50 (-)	50 (-)	50 (-)	50 (-)	50 (-)	50 (-)
24	25	47	0	47 (-)	47 (-)	47 (-)	47 (-)	47 (-)	47 (-)
25	7	10	4	12 (-)	12 (-)	12 (-)	12 (-)	12 (-)	12 (-)
26	19	9	28	28 (-)	28 (-)	28 (-)	28 (-)	28 (-)	28 (-)
27	13	22	0	22 (-)	22 (-)	22 (-)	22 (-)	22 (-)	22 (-)
28	161	70	123	184 (179)	184 (179)	182 (173)	188 (184)	182 (175)	183 (174)

29	137	83	73	151 (149)	151 (149)	156 (149)	160 (153)	156 (152)	167 (160)
30	103	72	0	122 (112)	122 (112)	126 (121)	118 (116)	116 (-)	124 (122)
31	94	94	0	117 (115)	117 (115)	114 (-)	121 (120)	120 (118)	120 (118)
32	78	81	0	100 (-)	100 (-)	103 (-)	105 (104)	103 (102)	97 (-)
33	211	114	106	250 (246)	250 (246)	252 (249)	254 (244)	241 (239)	249 (242)
34	164	94	0	189 (187)	189 (187)	200 (191)	194 (191)	191 (189)	188 (184)
35	124	123	0	145 (143)	145 (143)	160 (156)	150 (147)	154 (151)	151 (150)
36	90	111	0	124 (-)	124 (-)	135 (132)	129 (124)	134 (132)	133 (130)
37	81	81	0	100 (-)	100 (-)	101 (100)	110 (108)	107 (-)	109 (108)

H_ 1 - Heuristique idiote
H_2 - Heuristique de séparation de listes
H_3 - Heuristique de tri par consommation décroissante de ressources
H_4 - Heuristique de tri par consommation croissante de ressources
H_ 5 - Heuristique de tri par durée décroissante
H_6 - Heuristique de tri par durée croissante
RL - Recherche locale ("-" si pas d'amélioration)
En bleu : meilleur minorant et meilleur résultat sans recherche locale
En orange : meilleur résultat après recherche locale (si meilleur que sans recherche locale)

7.3. Graphiques des résultats



- Minorant
- Méthode 1 - Heuristique absurde
- Méthode 2 - Heuristique de séparation de liste
- Méthode 3 - Heuristique de tri de liste en fonction de la consommation décroissante de ressources
- Méthode 4 - Heuristique de tri de liste en fonction de la consommation croissante de ressources
- Méthode 5 - Heuristique de tri de liste par durée décroissante
- Méthode 6 - Heuristique de tri de liste par durée croissante
- Meilleure solution après recherche locale