Dossier scientifique

Résumé

Ceci est un dossier scientifique (donnant des explications sur des thèmes abordés dans l'exposition mais aussi sur des thèmes annexes). Nous avons donc repris les thématiques stand par stand de l'exposition.

Certaines explications sont indispensables. Par contre, d'autres sont clairement plus techniques et même si on conseille de les lire, on peut les passer dans un premier temps. Elles sont marquées par un  .

Table des matières

Chapitre 1

Algorithmes

1.1 Qu'est-ce qu'un algorithme ?

Le mot « algorithme » vient de Al Khwârizmî [1]. Un algorithme est une suite (finie) d'instructions claires permettant de résoudre un problème. C'est une façon systématique de procéder pour faire quelque chose. C'est un concept pratique, c'est un procédé systématique applicable mécaniquement, sans réfléchir, en suivant un mode d'emploi précis. Les instructions peuvent être très variées en fonction de l'agent à qui l'algorithme sera confié pour être réalisé, mais il est crucial qu'il n'y ait aucune ambiguïté dans celles-ci. C'est pourquoi, même s'il ne se réduit pas à son usage en informatique, l'algorithme est particulièrement adapté aux ordinateurs qui ont un jeu d'instructions bien défini (qui peut cependant dépendre de la machine), et qui sont là pour faire une suite d'opérations, très vite, sans réfléchir et sans erreurs.

1.2 Algorithmes de la vie courante

1.3 Algorithmes mathématiques

1.4 Algorithmes informatiques

1.5 Etude des algorithmes informatiques

1.6 La NP-complétude

1. mathématicien persan, $\sim$ 780-850. On lui doit notre système décimal de numération de position. Le mot algèbre vient aussi de lui, par l'intermédiaire de l'arabe al-jabr, opération consistant à basculer une soustraction d'un côté d'une équation en une addition de l'autre.

Chapitre 2

Bases de données

2.1 Les différents types de bases de données

2.2 Bases de données relationnelles

Chapitre 3

Ordonnancement

La recherche opérationnelle est l'ensemble des méthodes utilisées pour trouver la meilleure façon d'opérer en vue d'aboutir à un résultat visé, ou au meilleur résultat possible. Si on peut la faire remonter plus loin, on peut dire qu'elle prend son essor à la sortie de la seconde guerre mondiale, dans les années 50, surtout grâce à l'explosion de la capacité de calcul des ordinateurs.

La gestion de projets est une branche très importante de la recherche opérationnelle mais on peut y trouver plus généralement des problèmes de logistique, de planification, d'emploi du temps. Dans l'industrie manufacturière, les problèmes peuvent consister en trouver des plans de production (ordonnancement), diminuer le gaspillage des matières premières ou de l'énergie. Dans le domaine de la finance, il s'agit de maximiser le profit dans des investissements. Dans le domaine de l'informatique, on peut se demander où implanter des serveurs et avec quelle capacité de stockage de manière optimale, ou encore se demander dans quel ordre faire des tâches dans un système d'exploitation.

La recherche opérationnelle fait appel à plusieurs domaines mathématiques et informatiques : théorie des graphes, théorie des jeux, optimisation (non-)linéaire, intelligence artificielle.

Les problèmes de recherche opérationnelle sont souvent d'une grande complexité algorithmique.

3.1 Qu'est-ce qu'un problème d'ordonnancement ?

D'un point de vue mathématique, c'est un problème d'optimisation sous contraintes. Il s'agit de trouver, parmi des solutions réalisables (vérifiant les contraintes), la meilleure (qui minimise une ou plusieurs fonctions objectif). Plus précisément, un problème d'ordonnancement consiste à organiser dans le temps la réalisation de tâches compte tenu de contraintes temporelles et de contraintes sur les ressources disponibles.

Une **tâche** est une entité élémentaire dont la réalisation prend un certain temps et mobilise certaines ressources avec une certaine intensité. Parfois, on peut y ajouter une date à laquelle elle doit être terminée et inclure des pénalités de retard (à minimiser). Certaines tâches doivent être réalisées sans interruption (modèles non préemptifs), d'autres peuvent être effectuées par morceaux (modèle préemptif). Plusieurs tâches peuvent constituer une **activité** et plusieurs activités un **processus**.

Les **ressources** sont des moyens techniques ou humains disponibles en quantité limitée. Elles peuvent être renouvelables (moyens humains ou outils) ou consommables (argent, matière première). Parmi les ressources renouvelables, on peut distinguer celles qui sont disjonctives (qui ne peuvent exécuter qu'une tâche à la fois) et celles qui sont cumulatives (qui peuvent exécuter plusieurs tâches en même temps).

Les **contraintes** peuvent être temporelles : contraintes de temps alloué ou alors contraintes de successions des tâches. On a aussi des contraintes de ressources qui sont en nombre limité.

Les **objectifs** peuvent être liés au temps (finir à temps, le plus vite possible, etc.), aux ressources (en utiliser le moins possible). On peut essayer de trouver des solutions admissibles (qui finissent dans les temps par ex.) ou optimales (qui finissent le plus vite possible par ex.).

Si on se fonde sur un univers prédictible, on peut se contenter d'aller chercher la meilleure solution selon l'objectif. Mais si on veut tenir compte des perturbations potentielles, on va faire ce qu'on appelle de **l'ordonnancement robuste** : une bonne solution sera alors une solution qui ne se détériore pas trop s'il y a des perturbations. Il y a alors deux approches : proactives (qui anticipent les perturbations) et réactives (qui révisent en fonction des perturbations l'ordre dans lequel les tâches sont effectuées).

La solution d'un problème d'ordonnancement est souvent donnée sous forme de **diagramme de Gantt**.

3.2 Quelques exemples de problèmes d'ordonnancement

3.2.1 Problème de gestion de projet à contraintes de ressources

C'est un problème NP-difficile au sens fort [1]. Il consiste à ordonner un certain nombre de tâches entre lesquelles il y a des relations de précédence (une tâche ne peut débuter que si certaines tâches sont finies) et qui utilisent des ressources disponibles en quantité limitée. On a donc deux types de contraintes : les relations de précédence et la disponibilité des ressources. L'objectif est de finir le plus vite possible.

Si le diagramme de Gantt permet de visualiser avec une échelle de temps claire les dates de début et de fin des tâches, le diagramme PERT (*program evaluation and review technique*) est un outil d'analyse de ce type de problème. On représente l'ensemble des précédences par un graphe pour pouvoir déterminer les dates de début au plus tôt et au plus tard des différentes tâches. Il permet aussi de déterminer les tâches critiques qui sont celles qui retarderaient tout le projet en cas de délai.

Le PERT a été créé en 1958 à la demande de la Marine américaine pour planifier la durée de son programme de missiles balistiques nucléaires miniaturisés *Polaris*.

PERT sur un exemple simple [2]

Supposons que nous voulions organiser un événement qui requiert un certain nombre de tâches. On part du tableau de toutes les tâches avec leur durée de réalisation (et éventuellement les ressources utilisées [3]) et les relations de précédence :

1. A préciser, renvoyer à la section NP.
2. tiré de *Gestion de projet - réaliser le diagramme de PERT*, G. Casanova et D. Abécassis.
3. Nous ne tiendrons pas compte des ressources dans ce premier exemple.

	Nom de la tâche	Durée	Doit venir après
A	Définition du budget	4 jours	
B	Sélection thème, date, lieu	3 jours	A
C	Embauche du traiteur	3 jours	B
D	Annonce interne	3 jours	B
E	Annonce dans la presse	4 jours	D
F	Sélection menu	2 jours	C
G	Location des équipements	4 jours	C, E
H	Embauche personnel	4 jours	G
I	Préparatifs	5 jours	G
J	Événement	1 jour	F, H, I

Nous allons commencer par dresser une matrice des antériorités, ce qui aide pour tracer le graphe des précédences. Cette étape consiste à attribuer des niveaux à chaque tâche en fonction du nombre de tâches qui doivent être finies avant de commencer celle-ci. On va donc, pour chaque tâche, indiquer une croix lorsqu'elle doit venir après une autre et compter le total, ne garder que celles qui peuvent commencer directement (niveau 1) puis opérer par récurrence en éliminant du tableau les tâches déjà mises dans un niveau. Ici, on écrit

		il faut avoir terminé										**Niveaux**						
Pour faire		A	B	C	D	E	F	G	H	I	J	1	2	3	4	5	6	7
	A											0						
	B	x										1						
	C		x									1						
	D		x									1						
	E				x							1						
	F			x								1						
	G			x		x						2						
	H							x				1						
	I							x				1						
	J						x		x	x		3						
												A						

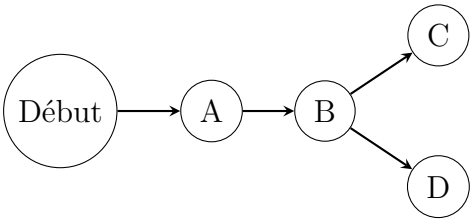
Seule la tâche A n'a aucune précédence, elle est donc de niveau 1. On l'élimine ensuite du tableau pour refaire les comptes pour le niveau 2 :

		il faut avoir terminé										**Niveaux**						
Pour faire		A	B	C	D	E	F	G	H	I	J	1	2	3	4	5	6	7
	A											0						
	B	x										1	0					
	C		x									1	1					
	D		x									1	1					
	E				x							1	1					
	F			x								1	1					
	G			x		x						2	2					
	H							x				1	1					
	I							x				1	1					
	J						x		x	x		3	3					
												A	B					

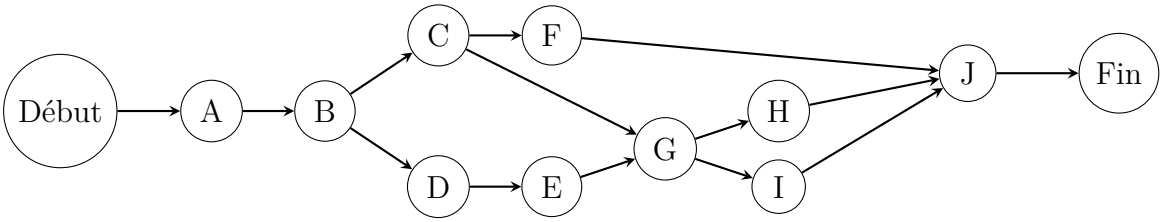
Et ainsi de suite pour déterminer tous les niveaux :

		il faut avoir terminé										Niveaux						
Pour faire		A	B	C	D	E	F	G	H	I	J	1	2	3	4	5	6	7
	A											0						
	B	x										1	0					
	C		x									1	1	0				
	D		x									1	1	0				
	E				x							1	1	1	0			
	F			x								1	1	1	0			
	G			x		x						2	2	2	1	0		
	H							x				1	1	1	1	1	0	
	I							x				1	1	1	1	1	0	
	J						x		x	x		3	3	3	3	2	2	0
												A	B	CD	EF	G	HI	J

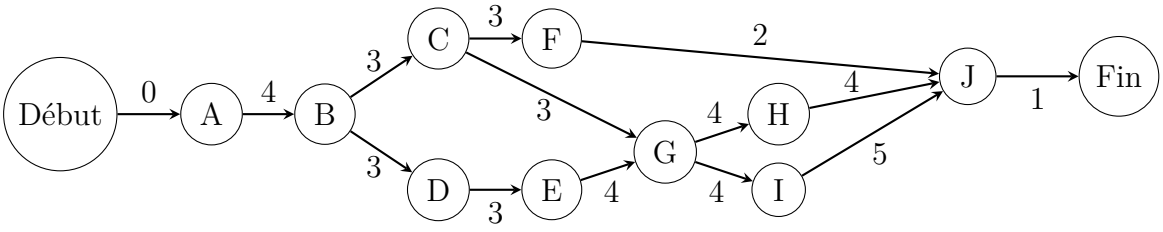
On va alors pouvoir commencer le graphe indiquant les précédences en y allant niveau par niveau. On commence par :



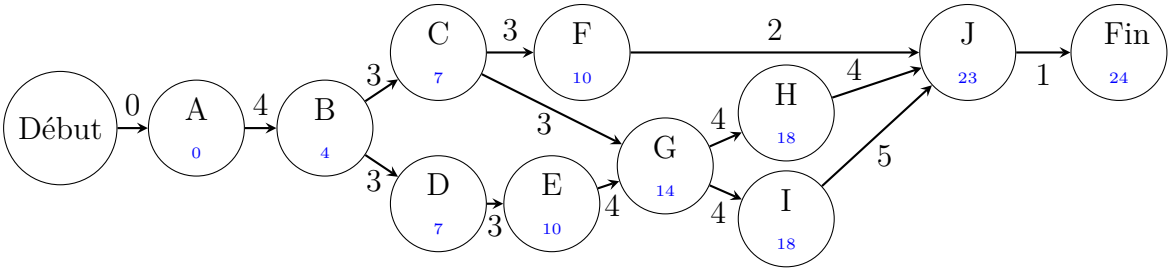
puis on va continuer en mettant une flèche entre deux tâches lorsqu'il y a précédence nécessaire :



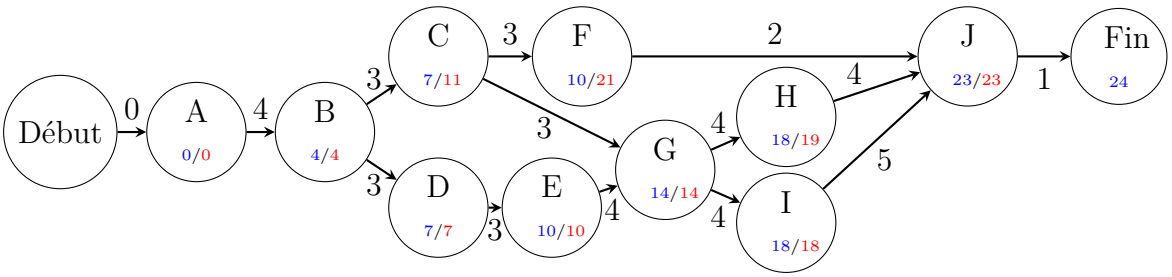
Il faut maintenant ajouter les temps de réalisation sur les flèches :



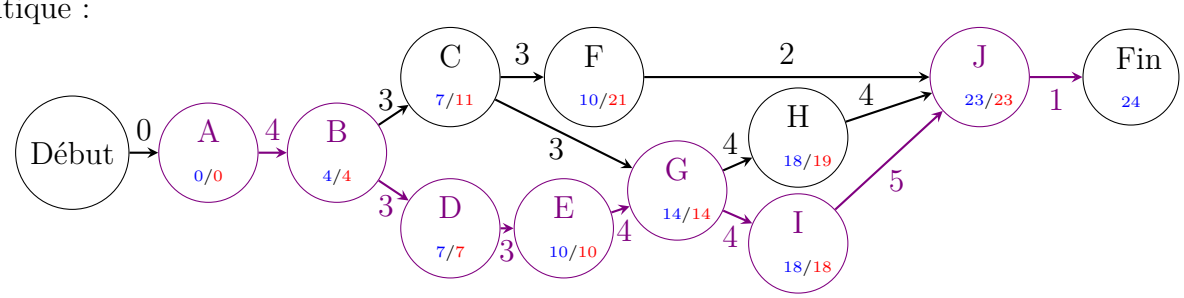
ce qui permet d'écrire les dates de démarrage au plus tôt de chaque tâche (en bleu) :



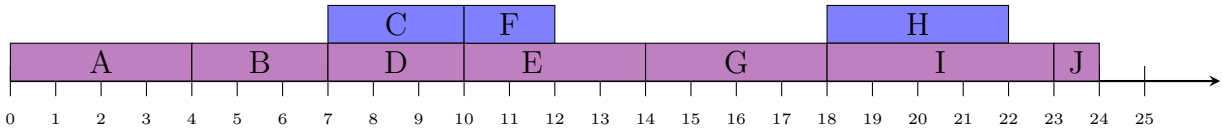
Le projet prendra donc 24 jours au minimum. On peut ensuite remonter le graphe pour déterminer, pour chaque tâche, la date de démarrage au plus tard (en rouge), qui est celle qui ne retardera pas la fin du projet :



Les tâches critiques sont celles qui ont une date de démarrage au plus tôt égale à la date de démarrage au plus tard, c'est-à-dire celles qui ne supportent aucun retard si on ne veut pas retarder la fin du projet. On peut les entourer en violet et cela trace également un chemin critique :



PERT permet donc une visualisation agréable des précédences et permet de trouver le temps minimal du projet avec les tâches critiques et les chemins critiques. C'est vraiment un outil d'analyse et on lui préfèrera un diagramme de Gantt pour le résultat final :



dans lequel on pourrait s'arranger pour indiquer les marges sur les tâches non critiques.

Un exemple avec des contraintes de ressources[4]

Peu importe la nature du projet, voici la liste des tâches avec leur durée et leur besoin en ressources humaines :

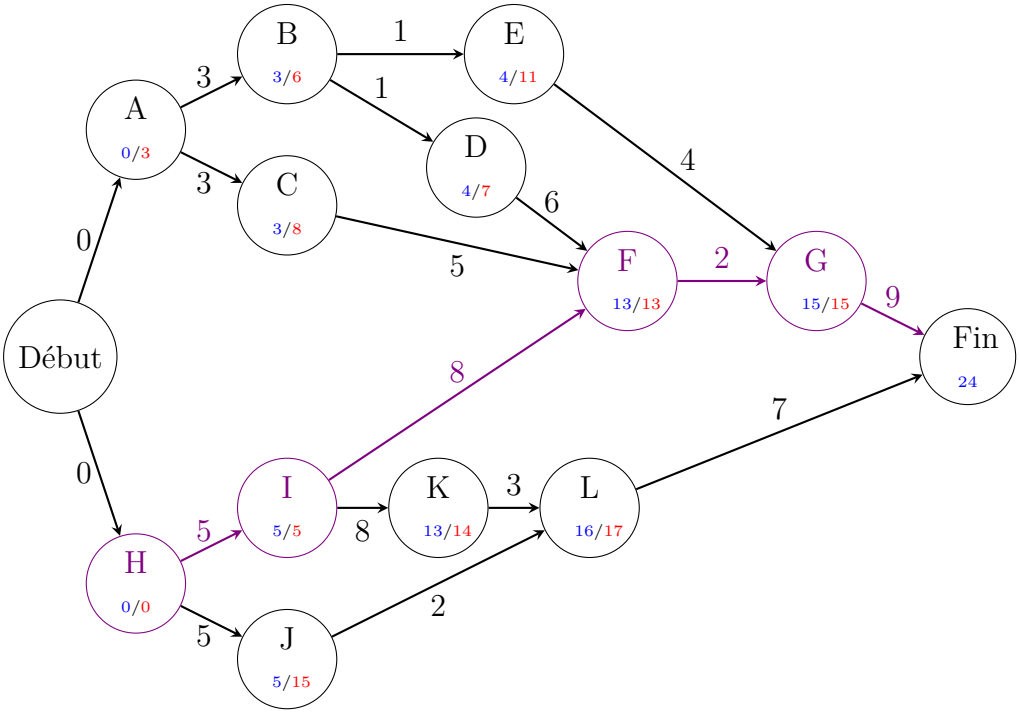
	Durée (en jours)	**Effectif (personnes)**	**Doit venir après**
A	3	5	
B	1	2	A
C	5	3	A
D	6	2	B
E	4	4	B
F	2	3	C, D, I
G	9	4	E, F
H	5	4	
I	8	4	H
J	2	2	H
K	3	2	I
L	7	4	J, K

On commence par écrire la matrice des antériorités :

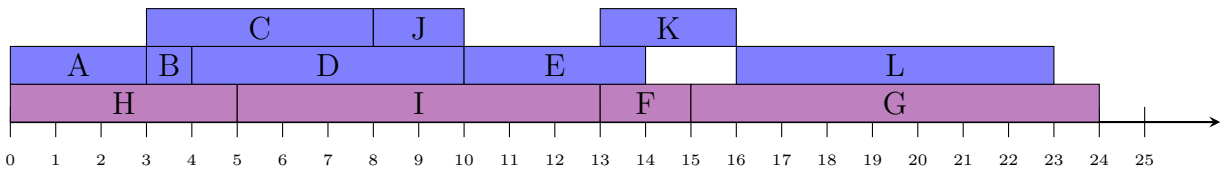
4. tiré de *Gestion de projet - réaliser le diagramme de PERT*, G. Casanova et D. Abécassis.

		il faut avoir terminé												Niveaux				
Pour faire		A	B	C	D	E	F	G	H	I	J	K	L	1	2	3	4	5
	A													0				
	B	x												1	0			
	C	x												1	0			
	D		x											1	1	0		
	E		x											1	1	0		
	F			x	x					x				3	3	1	0	
	G					x	x							2	2	2	1	0
	H													0				
	I								x					1	0			
	J								x					1	0			
	K									x				1	1	0		
	L										x	x		2	2	1	0	
														AH	BCIJ	DEK	FL	G

pour pouvoir faire le graphe PERT :



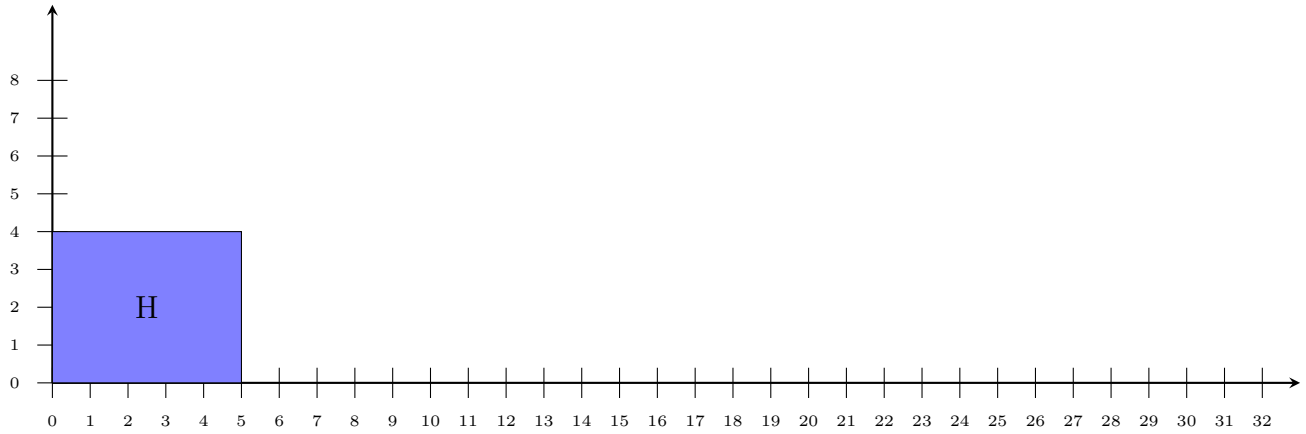
Cela nous donne le temps minimal, les tâches critiques et le chemin critique, les marges sur les différentes tâches. On peut ensuite faire un diagramme de Gantt et essayer de voir de quel effectif nous aurons besoin (en imaginant que tout le monde est polyvalent) pour mener le projet à bien en 24 jours.



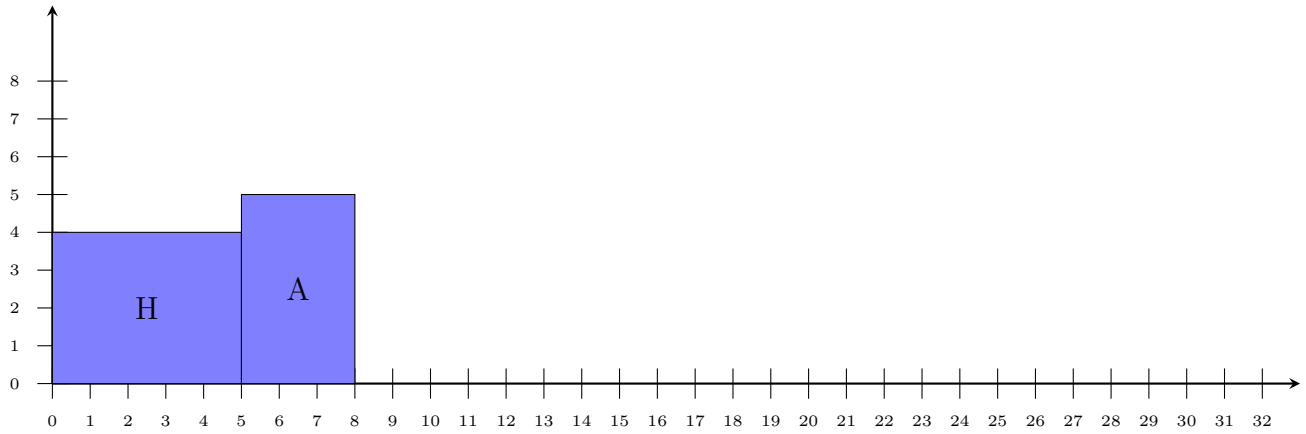
Une équipe de 9 personnes suffira, sachant qu'on ne peut pas faire moins sans retarder le projet à cause des tâches A et H qui doivent se chevaucher. La méthode PERT est particulièrement bien adaptée à **l'ordonnancement de projet** qui ne tient compte que des relations de précédence entre les tâches mais pas des ressources : soit elles sont illimitées, soit elles sont suffisantes pour ne pas être une contrainte. Maintenant, si on a seulement une équipe de 8 personnes, cela

complique l'analyse et il est plus difficile de trouver la solution optimale. Il faut alors utiliser l'axe des ordonnées pour représenter les ressources et en mettant de l'épaisseur aux tâches. Un algorithme heuristique, parmi d'autres, consiste à utiliser le travail fait précédemment et à essayer de donner un ordre de priorité aux tâches. Par exemple, on regarde les tâches de niveau 1, on choisit de commencer par celle qui demande le plus de ressources puis, parmi les tâches libérées, on regarde lesquelles on peut démarrer par ordre de priorité. Pour l'ordre de priorité, on peut en choisir plusieurs, celle qui a le plus de retard par rapport à la date de démarrage au plus tard, celle qui demande le plus de ressources, etc. Rien ne donnera l'optimal à tous les coups. Essayons ici de donner la priorité aux tâches les plus en retard (ou les moins en avance) :

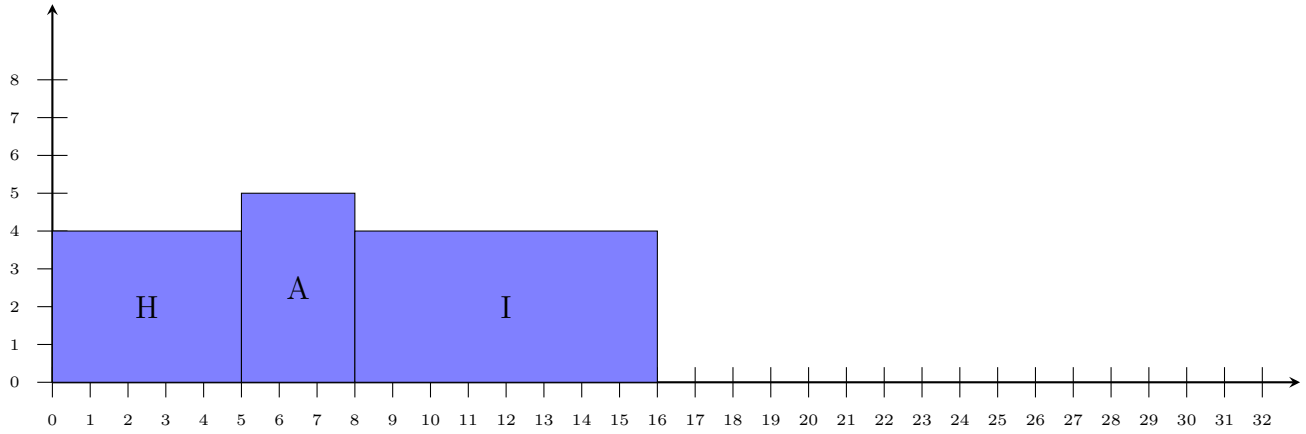
- On peut commencer par A et H et on commence par H, le plus urgent.



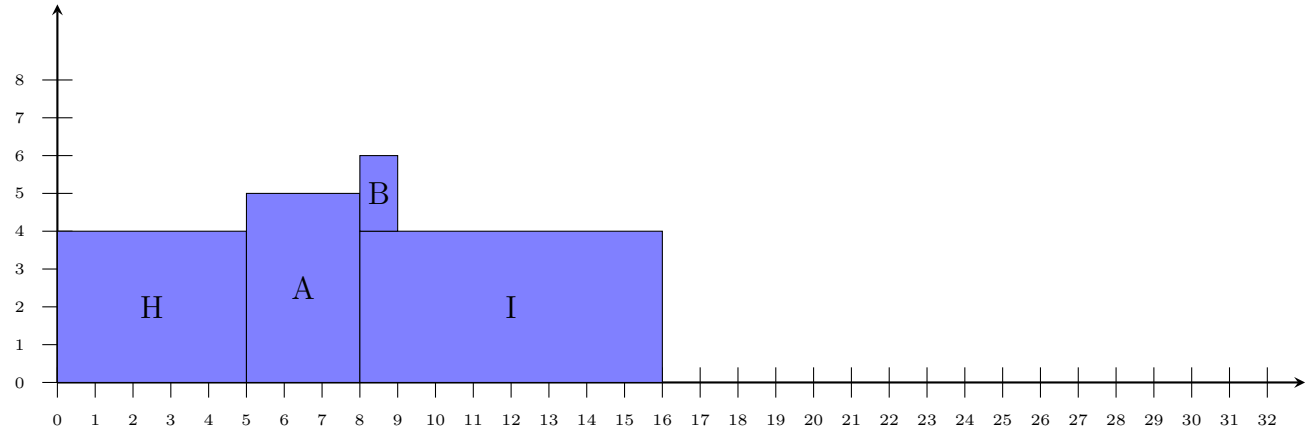
- A, I et J sont libres. On choisit A qui est en retard.



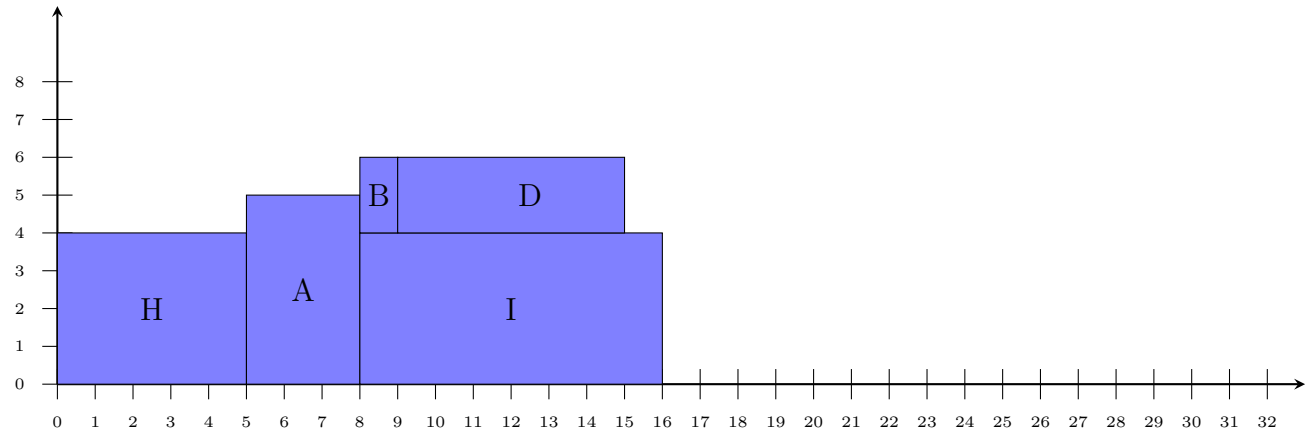
- B, C, I, et J sont libres. On choisit I qui est le plus en retard.



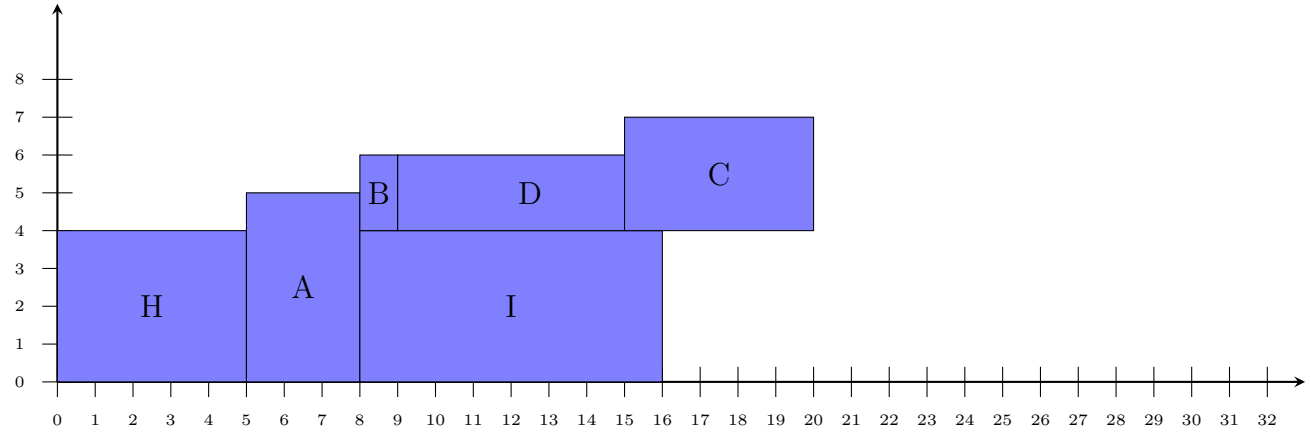
- B, C, J et K sont libres. On choisit B qui est à démarrer d'abord.



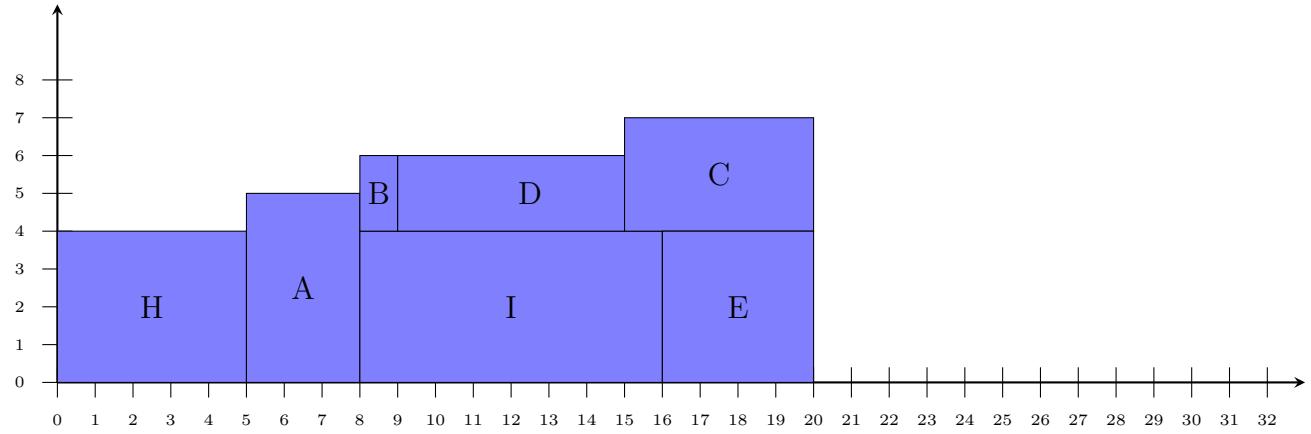
- C, D, E, J et K sont libres. On choisit D qui est à démarrer d'abord.



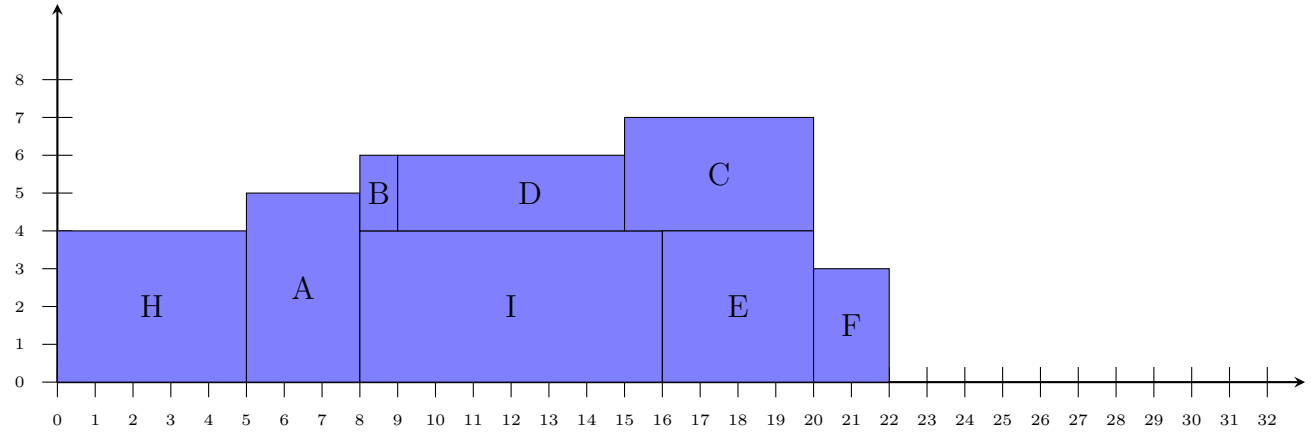
- C, E, J et K sont libres. On choisit C qui est à démarrer d'abord.



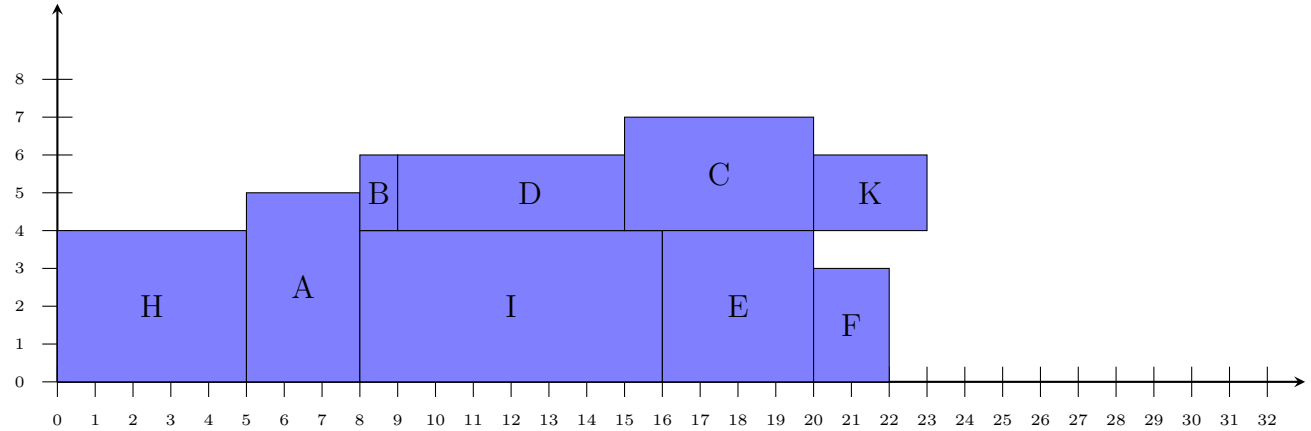
- E, F, J et K sont libres. On choisit E qui est à démarrer d'abord.



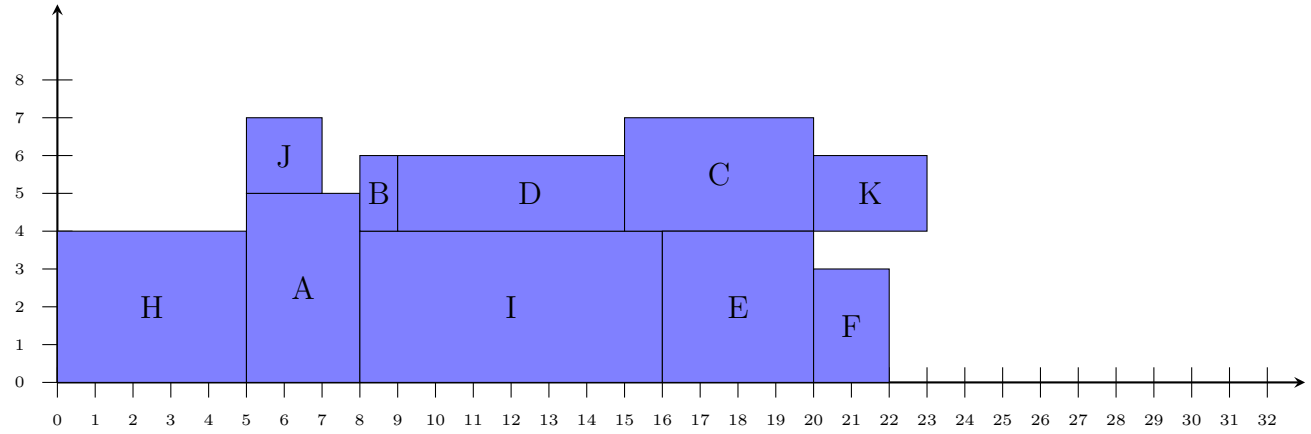
- F, J et K sont libres. On choisit F qui est à démarrer d'abord.



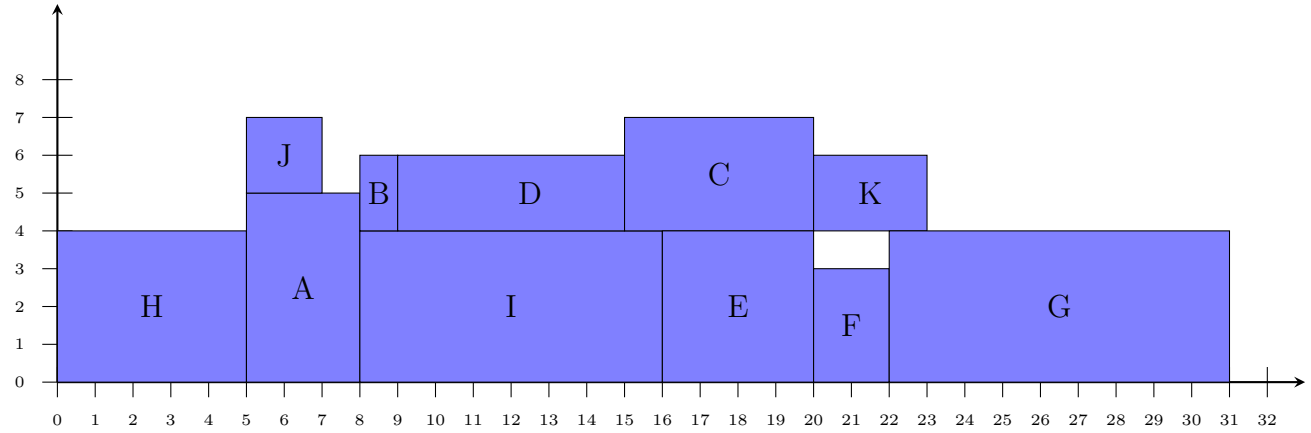
- G, J et K sont libres. On choisit K qui est à démarrer d'abord.



- G et J et L sont libres. On choisit J qui est à démarrer d'abord.

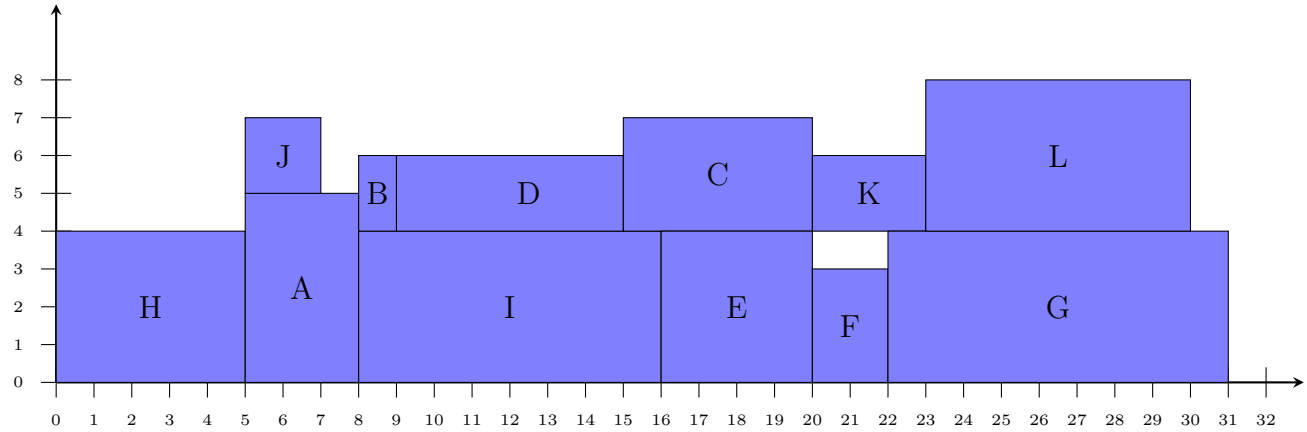


- G et L sont libres. On choisit G qui est à démarrer d'abord.



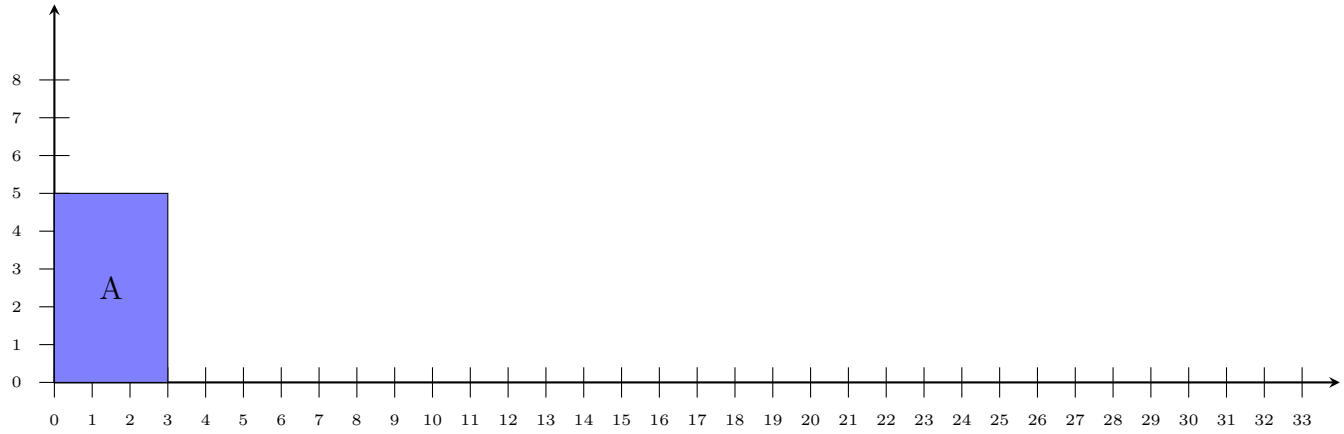
- On termine par L.

Cela donne la solution admissible suivante :

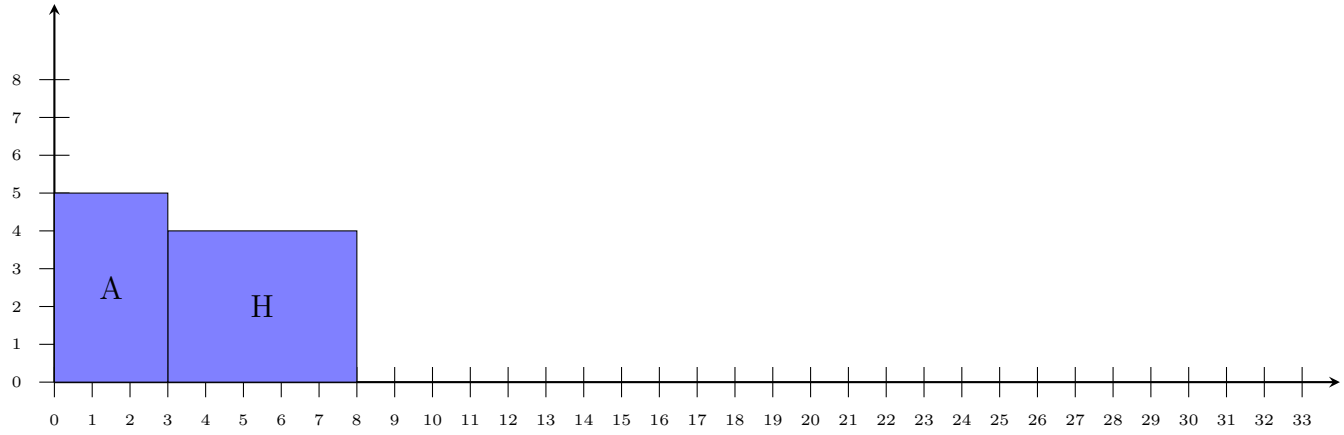


Mais on aurait également pu choisir de donner la priorité aux tâches demandant le plus de personnel, et en cas d'égalité à la plus en retard (ou la moins en avance). Cela change un peu l'ordre :

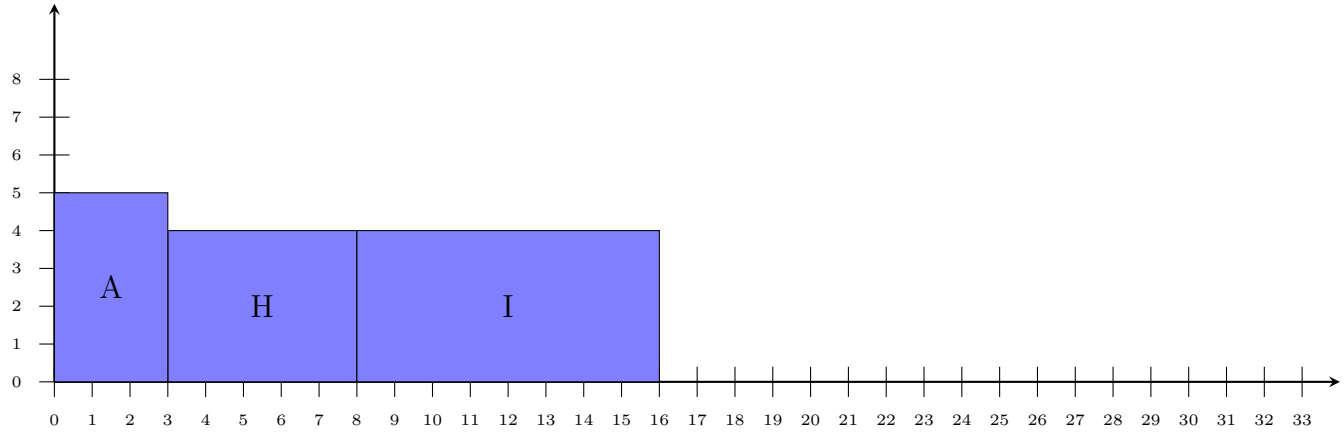
- On peut commencer par A et H et on commence par A, qui demande 5 personnes.



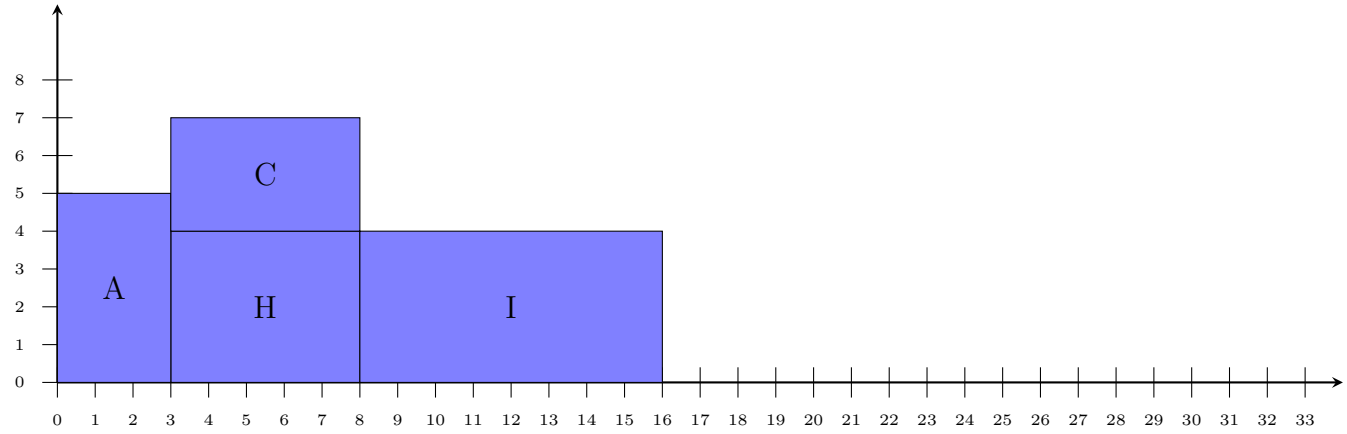
- B, C et H sont libres. On choisit H qui demande 4 personnes.



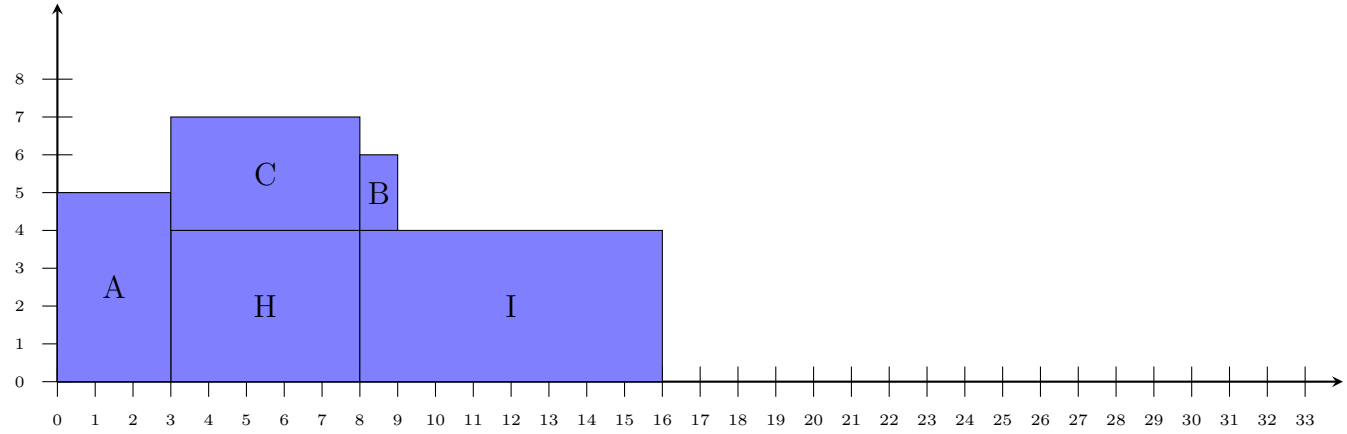
- B, C, I, et J sont libres. On choisit I qui demande 4 personnes.



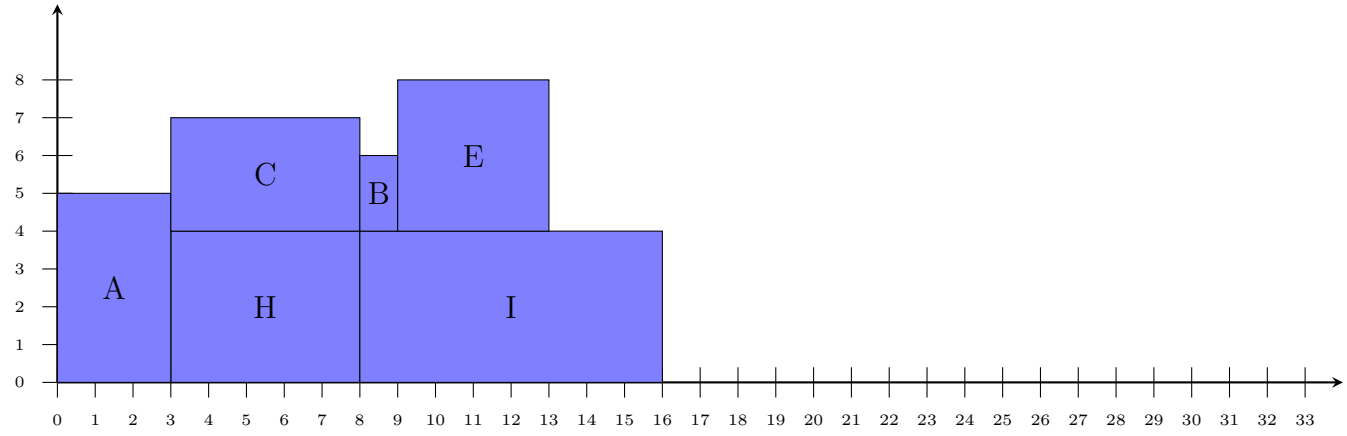
- B, C, J et K sont libres. On choisit C qui demande 3 personnes.



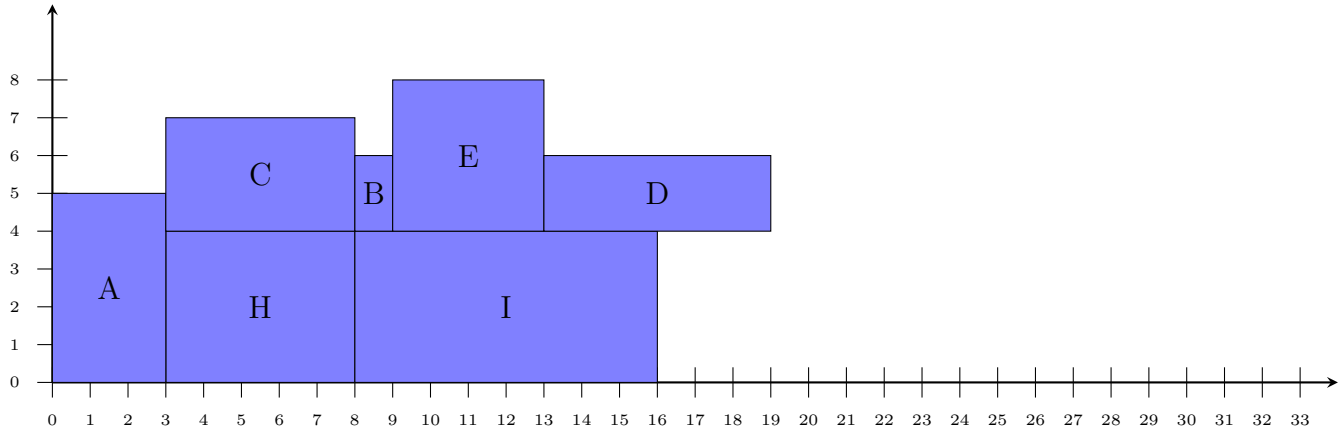
- B, J et K sont libres. Toutes demandent 2 personnes, on choisit B qui doit démarrer le plus tôt.



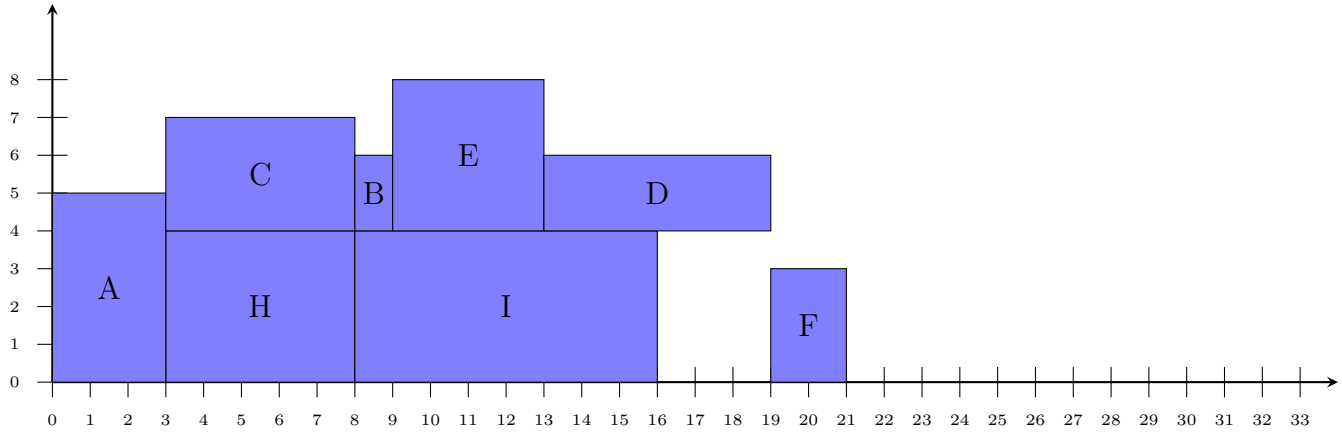
- D, E, J et K sont libres. On choisit E qui demande 4 personnes.



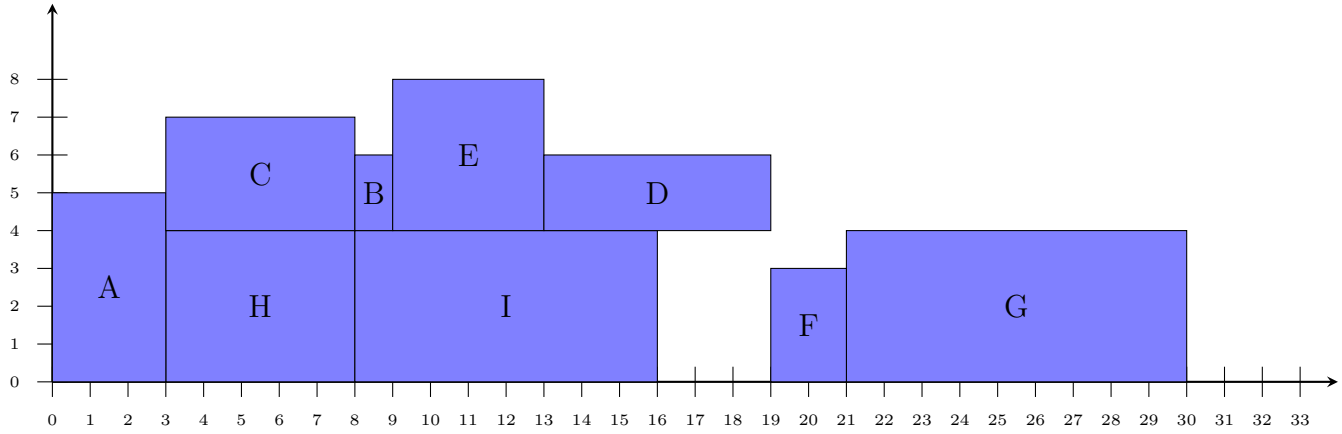
- D, J et K sont libres. Toutes demandent 2 personnes, on choisit D qui doit démarrer le plus tôt.



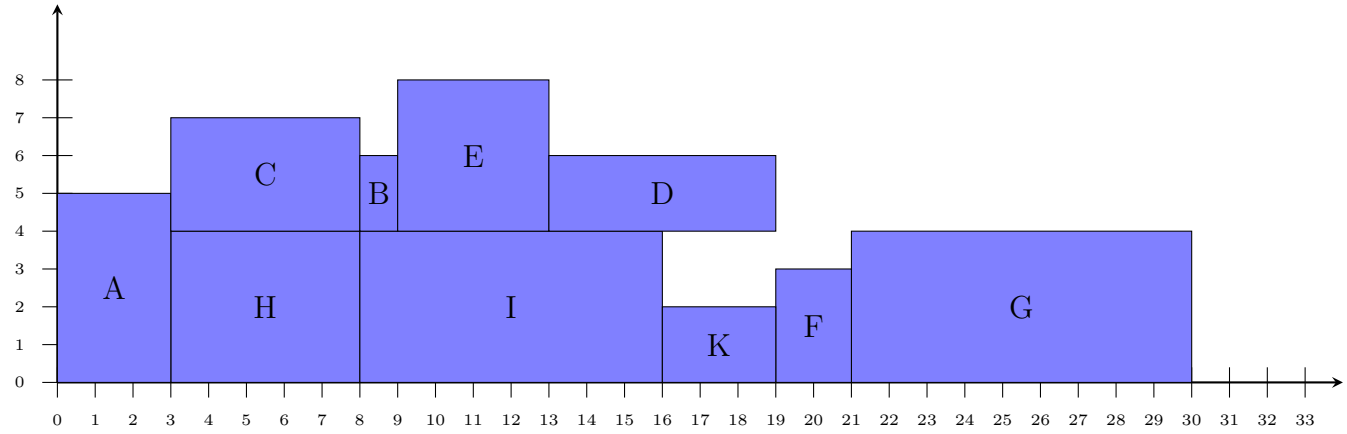
- F, J et K sont libres. On choisit F qui demande 3 personnes.



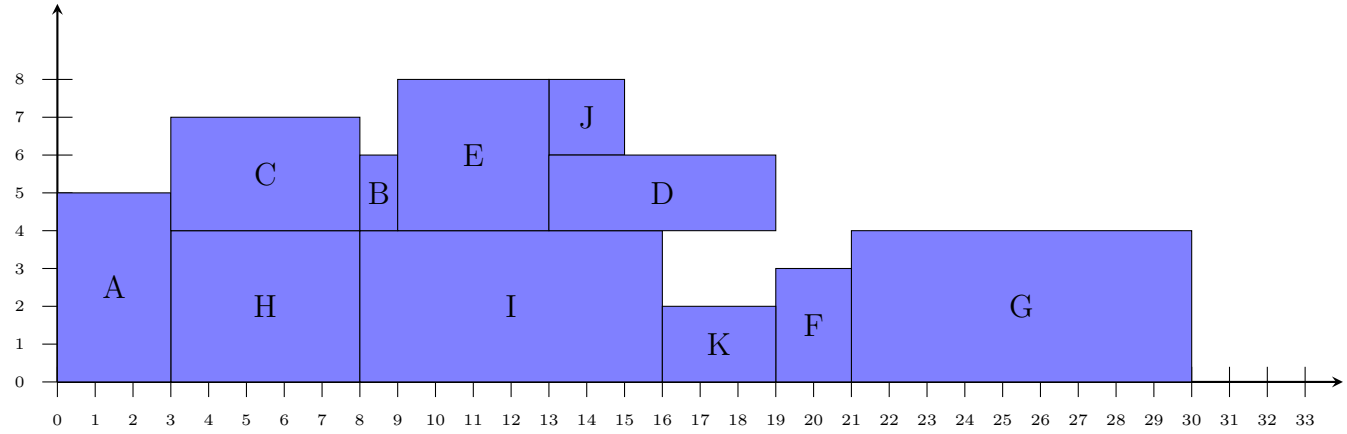
- G, J et K sont libres. On choisit G qui demande 4 personnes.



- J et K sont libres. Toutes demandent 2 personnes, on choisit K qui doit démarrer le plus tôt.

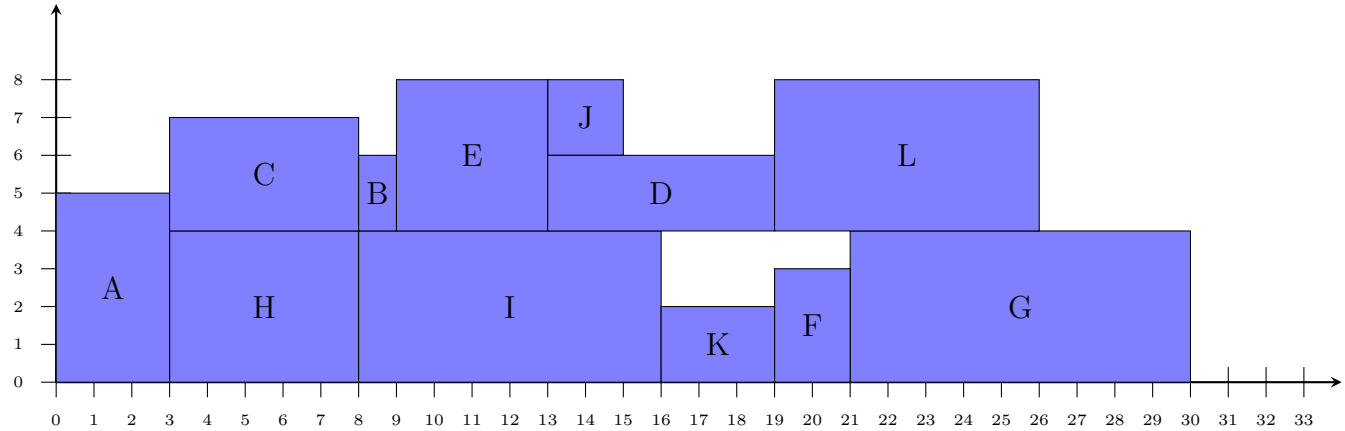


- J seul est libre. On choisit donc J.



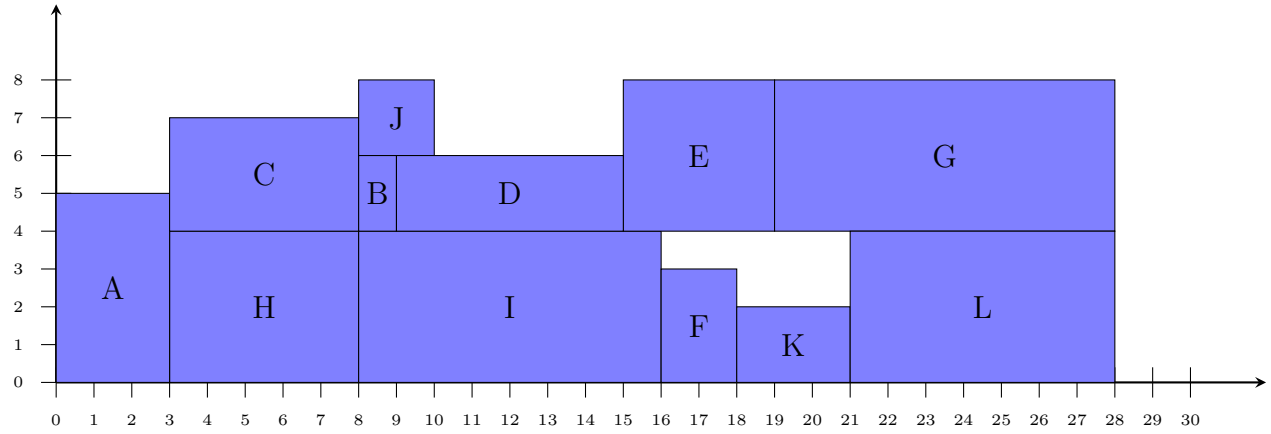
- On termine par L.

Cela donne la solution admissible suivante :



qui fait mieux que la précédente.

Mais, en fait, on peut faire mieux. On ne pourra sûrement pas faire moins de 27 jours (temps minimal + tâche A à caser) mais voici une solution en 28 jours :



Peut-on faire en 27 jours ? Le seul endroit où on perd un jour est entre les tâches F et G. Pourrait-on se débrouiller pour faire commencer la tâche G un jour avant ? On se rend compte que non car il faudra bien faire C, E et la séquence B-D avant G et on ne peut pas les faire en même temps, faute de moyens humains.

On voit ici toute la difficulté du problème dès que les contraintes s'accumulent. De plus, nous n'avons pris comme objectif que le temps total du projet mais nous pourrions aussi essayer de minimiser les coûts salariaux en imaginant que nous sommes obligés d'employer le nombre de personnes utilisées sur toute la durée du projet. Cela reviendrait à essayer de mettre toutes ces tâches en vérifiant les contraintes de précédence dans le plus petit rectangle possible. Par exemple, cette dernière solution ($28 \times 8 = 224$ jours $\times$ personne) est moins bonne que la toute première ($24 \times 9 = 216$ jours $\times$ personne).

Algorithmes utilisés

Ils sont nombreux mais on peut noter :

- optimisation linéaire : des machines ne sont utilisables qu'un certain nombre d'heures par mois et on veut fabriquer différents produits qui utilisent ces machines et seront vendus un certain prix. Comment maximiser le prix de vente ? Cela peut se traduire en un problème d'optimisation linéaire [5]. Les algorithmes servant à résoudre ce type de problème sont ceux du simplexe ou des points intérieurs par exemple.L'optimisation linéaire est utile dans les problèmes d'optimisation à moyen et long terme (problèmes stratégiques et tactiques en recherche opérationnelle) et sert dans des domaines variés : industrie manufacturière, finance, transports, télécommunication.

- programmation par contraintes : l'idée est ici de séparer les contraintes (problème de satisfaction de contraintes) de la recherche d'optimum dans laquelle on utilisera les contraintes du problème afin de réduire la taille de l'espace des solutions possibles à parcourir. Très souvent, les algorithmes propagent les contraintes afin de réduire le nombre de candidats à explorer (on peut penser à un arbre à parcourir dans lequel la propagation des contraintes va permettre d'élaguer certaines branches). Un bon exemple de programmation par contraintes, qui n'a certes rien à voir avec l'ordonnancement, est la résolution de sudoku.

- algorithme par séparation et évaluation (*branch and bound*) : lorsqu'on cherche un minimum parmi un ensemble trop grand de possibilités, cet algorithme heuristique va évaluer les noeuds d'un arbre soit en trouvant l'optimum sur toutes les branches partant de ce noeud soit en montrant que l'optimum sur ces branches sera moins bon que ce qu'on peut faire (pour faire cela, on relaxe certaines contraintes, cela rendant le calcul plus simple et

5. Voir annexe ?

donnant des bornes sur le résultat). Cela évite ensuite de parcourir les branches à partir de ce noeud. Il faut aussi parcourir intelligemment l'arbre pour essayer de faire apparaître le meilleur résultat en premier (ou un bon), cela permet d'élaguer plus vite le reste.

- et bien d'autres en fonction des problèmes bien sûr.

Faut-il faire une annexe pour détailler ces grandes classes d'algorithmes ?

3.2.2 Ordonnancement dans les systèmes d'exploitation

Dans un système d'exploitation, l'ordonnanceur (*scheduler* en anglais) est le composant du noyau chargé de choisir l'ordre d'exécution des processus sur les processeurs d'un ordinateur ou d'un téléphone. Aujourd'hui, la plupart des processeurs sont multi-cœurs. En effet, pour augmenter la vitesse de calcul, on peut augmenter la fréquence du processeur mais l'énergie dissipée est proportionnelle au carré de la fréquence : multiplier la fréquence par 2 revient à multiplier la puissance dissipée par 4. Cela finit par poser des problèmes de refroidissement, surtout lorsqu'on cherche à miniaturiser les ordinateurs ou les téléphones. Ainsi, il est plus efficace de multiplier les cœurs sur un même processeur : multiplier par 2 le nombre de cœurs ne multiplie que par 2 la puissance dissipée. Un cœur est un ensemble capable d'exécuter des calculs de manière autonome. Aujourd'hui, il est standard d'avoir des processeurs avec 4 cœurs mais beaucoup en ont huit ou plus maintenant [6]. Certains cœurs sont efficaces pour certaines tâches, d'autres pour d'autres.

Il n'est pas rare sur un ordinateur ou un téléphone de devoir effectuer plus de tâches que le processeur ne peut en prendre en charge simultanément. Le système appelle alors régulièrement l'ordonnanceur qui va élire la prochaine tâche à effectuer. On peut prendre en compte plusieurs paramètres en donnant la priorité à certaines tâches (en particulier celles qui sont interactives), en essayant de tout terminer dans les temps, etc. Par exemple, il est judicieux de traiter les tâches cruciales pour le fonctionnement en priorité, avant d'envoyer votre message. Car, si votre téléphone crashe, le message ne sera de toute façon pas envoyé.

Le choix de l'algorithme d'ordonnancement est important puisqu'il conditionne le comportement du système. Deux grandes classes existent : **l'ordonnancement en temps partagé** qui donnera la priorité aux tâches interactives par exemple ; **l'ordonnancement en temps réel** utile dans les systèmes embarqués, qui s'assure que les tâches sont terminées dans un délai donné.

Les algorithmes d'ordonnancement utilisés dans les systèmes d'exploitation sont nombreux. Le problème étant compliqué et aucun algorithme optimal ne fonctionnant en temps raisonnable, on voit ici l'intérêt d'avoir des algorithmes donnant des bonnes solutions rapidement : si l'ordonnanceur met une seconde à choisir quelle tâche faire, on aurait mieux fait de les faire dans le désordre, sans réfléchir, cela aurait été plus vite.

3.2.3 Ordonnancement d'ateliers

L'objectif est d'organiser dans le temps diverses tâches pour utiliser au mieux les ressources disponibles et finir dans les temps ou le plus vite possible.

Une **tâche** est une entité élémentaire prenant un temps p_i, donc commençant au temps t_i et terminant au temps $c_i = t_i + p_i$. Celle-ci consomme dans cet intervalle de temps des ressources k avec une charge a_{ik}. Bien sûr, certaines tâches peuvent être exécutées par morceaux (modèles préémptifs), d'autres non (modèles non préemptifs), la consommation en ressources n'est pas nécessairement constante au cours de l'exécution mais le problème étant déjà suffisamment

6. Mon ordinateur portable en possède 10 et mon téléphone 6 [l'auteur de ces lignes]

compliqué comme cela, dans la plupart des modèles, on supposera cette charge constante (quitte à découper la tâche en deux ou trois si ce n'est pas le cas de manière significative).

Une **ressource** k est un moyen humain, technique ou financier, disponible en quantité limitée (A_k). Une ressource peut être renouvelable mais utilisable en quantité limitée à chaque instant (ressources humaines, machines, etc.), elle peut aussi être consommable et c'est alors sa quantité globale qui est limitée (matières premières, argent, etc.).

Les **contraintes temporelles** sont de plusieurs types :

- contraintes de précédence (ordre dans lequel certains tâches doivent être effectuées). Cela se traduit par des inégalités du type $t_j \geq c_i$ (il faut avoir fini la tâche i pour commencer la tâche j).
- les contraintes de délai du type $c_i \leq T_i$, la tâche i doit être terminée avant le temps T_i sous peine de pénalité de retard.

Les **contraintes de ressources** dépendent de leur nature cumulative ou disjonctive :

- Pour les ressources disjonctives, si deux tâches utilisent la même ressource, on le traduira par le fait que $t_j \geq c_i$ ou $t_i \geq c_j$ (une tâche ne peut commencer que quand l'autre a terminé).
- Pour les ressources cumulatives, on regardera à tout temps t toutes les tâches étant exécutées à cet instant et utilisant la ressource k, i.e. tous les $i \in T_k(t)$, c'est-à-dire avec $t_i \leq t < c_i$ et $a_{ik} > 0$. Et la contrainte sera que, pour tout t, $\sum_{i \in T_k(t)} a_{ik} \leq A_k$.

On peut distinguer dans l'industrie manufacturière plusieurs types de problèmes d'ordonnancement d'ateliers :

Machine unique

Toutes les tâches doivent passer un certain temps p_i sur une machine unique. Cela peut paraître absurde et simple mais c'est le cas lorsqu'il y a dans une chaîne de productions un goulot d'étranglement, une machine par laquelle toutes les tâches doivent passer et que l'on possède en un seul exemplaire. Bien souvent, dans ce cas, tout va dépendre de l'ordonnancement sur cette machine. On peut par exemple imaginer le problème suivant : chaque tâche i peut arriver au plus tôt au temps T_i sur cette machine (car la production avant le passage sur cette machine demande un temps T_i) et devra subir encore C_i de temps de travail après passage sur cette machine. Le problème est dans quel ordre tout faire sur cette machine afin que tout finisse le plus vite possible, i.e. minimiser $max_i\,(t_i + p_i + C_i)$ sous les contraintes $t_i \geq T_i$ et $(t_j \geq t_i + p_i) \vee (t_i \geq t_j + p_j)$ pour tout couple de tâches distinctes.

Machines parallèles

C'est exactement le même problème que ci-dessus mais on possède plusieurs (k) machines du même type, on peut donc faire un maximum de k tâches simultanément.

Ateliers à cheminement unique (flow shop)

Tous les produits sont élaborés de la même façon, ils sont tous constitués d'un certain nombre de tâches, toujours exécutées dans le même ordre, et utilisant une machine à chaque fois (les machines peuvent être uniques ou en multiples exemplaires). La durée des tâches est variable. L'objectif est souvent de finir dans les temps ou le plus vite possible. Dans ce cadre, à une tâche correspond une machine (le plus souvent), une activité est constituée de plusieurs

tâches mais toutes les activités passent par les mêmes machines (avec des temps différents cependant). Ceci s'applique à beaucoup d'exemples : industrie textile, automobile, etc.

Dès qu'il y a plus de deux machines, le problème devient NP-difficile. Ce type d'ordonnancement est détaillé dans les sections suivantes.

Ateliers à cheminement multiple (job shop)

Les différents produits (activités) requièrent différentes tâches à faire dans un certain ordre et passant sur plusieurs machines successivement. La différence avec le *flow shop* est que toutes les activités ne passent par sur les mêmes machines et dans le même ordre. L'objectif est bien souvent également de minimiser le temps total de production. Ce problème est considéré parmi les plus difficiles à traiter. Voici un exemple avec 4 activités et 6 machines :

- L'activité 1 doit passer successivement sur les machines 1 (6 unités de temps), 4 (3 unités de temps), 3 (8 unités de temps).
- L'activité 2 doit passer successivement sur les machines 2 (4 unités de temps), 3 (7 unités de temps) et 5 (5 unités de temps).
- L'activité 3 doit passer successivement sur les machines 3 (8 unités de temps), 2 (3 unités de temps) et 6 (5 unités de temps).
- L'activité 4 doit passer successivement sur les machines 1 (4 unités de temps), 2 (9 unités de temps), 5 (6 unités de temps) et 6 (7 unités de temps).

En combien de temps peut-on finir toutes les activités ?

Les exemples concernent des usines produisant plusieurs types de produits avec plus ou moins les mêmes ressources.

Autres modèles

Il existe bien entendu d'autres modèles dans l'industrie manufacturière : flow shop avec certaines machines parallèles, open shop (ordre des tâches libre au sein d'une activité), etc.

3.3 Le problème du flow-shop

Dans le problème du flow-shop, on a un certain nombre d'activités (pour $i = 1, \ldots, n$) constituées de tâches successives (au nombre de m) qui doivent passer sur m machines distinctes. Pour chaque activité $i \in \{1, \ldots, n\}$, on notera $T_{i,j}$ son temps de passage sur la machine $j \in \{1, \ldots, m\}$ (le temps de la tâche j de l'activité i). On notera également $t_{i,j}$ le temps de début de l'activité i sur la machine j. On a donc les contraintes suivantes :

- **Contrainte de précédence** : pour tout $i \in \{1, \ldots, n\}$ et tout $j \in \{1, \ldots, m-1\}$,

$$t_{i,j+1} \geq t_{i,j} + T_{i,j} \,, \tag{3.1}$$

  ce qui traduit le fait que l'activité i ne peut démarrer sur la machine $j+1$ qu'une fois qu'elle est terminée sur la machine j.

- **Contrainte de ressource** (machines uniques) : pour tout $k \in \{1, \ldots, m\}$, pour tous $i, j \in \{1, \ldots, n\}$ avec $i \neq j$,

$$(t_{i,k} + T_{i,k} \leq t_{j,k}) \vee (t_{j,k} + T_{j,k} \leq t_{i,k}) \ . \tag{3.2}$$

La question est donc de déterminer l'ordre de passage des activités sur chaque machine (on commencera alors toujours au plus tôt possible en fonction de ces ordres) afin d'atteindre un objectif : finir avec un temps total inférieur ou égal à une valeur donnée, minimiser le temps total, ou alors parfois minimiser la somme des temps de finition des diverses activités sur la dernière machine.

Dans la suite, on cherchera toujours à minimiser le temps de fin de la dernière activité sur la dernière machine.

A noter que pour $m = 1$, le problème n'a aucun intérêt, tous les ordres de passage donnant le même temps final.

3.3.1 Avec ou sans permutations ?

Le problème du flow-shop peut se décomposer en deux problèmes distincts, selon qu'on admette les permutations d'ordre de passage des activités entre deux machines ou que l'ordre de passage sur la première machine soit celui de toutes les machines. On appelle *permutation flow-shop* le problème où l'ordre de passage est le même sur toutes les machines et *standard flow-shop* celui, plus général, où on permet des changements d'ordre entre machines.

Grâce au théorème suivant, ces deux problèmes sont équivalents tant qu'il y a 3 machines ou moins :

Théorème 3.1. *Considérons le problème du standard flow-shop dans lequel il y a n activités et m machines et pour lequel l'objectif est de minimiser le temps de fin de la dernière activité sur la dernière machine. Alors il n'y a aucun intérêt à changer l'ordre de passage entre les machines 1 et 2 et il n'y a aucun intérêt à changer l'ordre de passage entre les machines $m-1$ et m.*

Preuve - Commençons par montrer qu'il ne sert à rien de changer l'ordre de passage entre l'avant-dernière et la dernière machine. Imaginons que l'ordre de passage sur l'avant-dernière machine soit $1, 2, 3, \ldots, n$ et que l'ordre de passage sur la dernière machine soit $(\sigma(i))_{i=1,\ldots,n}$ avec σ une permutation de $\{1, \ldots; n\}$. On va montrer que cette permutation peut être supposée égale à l'identité sans augmenter le temps total d'exécution. Soit $1 \leq i \leq n-1$, supposons que $j = \sigma(i) > \sigma(i+1) = k$. Il existe nécessairement un tel i sauf si $\sigma = Id$. La tâche k ayant été terminée avant la tâche j sur l'avant-dernière machine puisque $k < j$, on peut inverser sans problème les deux tâches j et k sur la dernière machine sans augmenter le temps de démarrage. En effet, on avait $t_{k,m-1} + T_{k,m-1} \leq t_{j,m-1} \leq t_{j,m-1} + T_{j,m-1} \leq t_{j,m}$ donc on peut bien faire démarrer la tâche k au temps $t_{j,m}$ voire avant sur la dernière machine. Cela n'augmentera pas le temps total d'exécution des deux tâches j et k sur la dernière machine et donc n'augmentera pas le temps total d'exécution. Ainsi on peut sans augmenter le temps total d'exécution modifier σ de sorte à ce que $\sigma(i) < \sigma(i+1)$. Comme on peut le faire pour tout $1 \leq i \leq n-1$, on peut en fait supposer que $\sigma = Id$ sans augmenter le temps total d'exécution (voire en le diminuant).

Montrons maintenant qu'il ne sert à rien de changer l'ordre de passage entre la première et la deuxième machine. Le raisonnement est assez symétrique. Imaginons que l'ordre de passage sur la deuxième machine soit $1, 2, 3, \ldots, n$ et que l'ordre de passage sur la première machine soit $(\sigma(i))_{i=1,\ldots,n}$ avec σ une permutation de $\{1, \ldots; n\}$. On va montrer que cette permutation peut être supposée égale à l'identité sans augmenter le temps total d'exécution. Soit $1 \leq i \leq n-1$, supposons que $j = \sigma(i) > \sigma(i+1) = k$. Il existe nécessairement un tel i sauf si $\sigma = Id$. On a alors $t_{j,2} \geq t_{k,2} \geq t_{k,1} + T_{k,1}$. Inverser les deux tâches sur la première machine est toujours possible, ne changera pas le temps de fin d'exécution de ces deux tâches prises ensemble et cela n'affectera pas le temps de démarrage des tâches sur la deuxième puisque la tâche k pourra commencer au temps $t_{k,2}$ et la tâche j au temps $t_{j,2} \geq t_{k,1} + T_{k,1}$ qui est le nouveau $t_{j,1} + T_{j,1}$.

Ainsi on peut supposer que $\sigma(i) < \sigma(i+1)$ sans augmenter le temps total d'exécution. Comme on peut le faire pour tout $1 \leq i \leq n-1$, on peut en fait supposer que $\sigma = Id$ sans augmenter le temps total d'exécution (voire en le diminuant). $\diamondsuit$

Corollaire 3.1. *Pour $m \leq 3$, les permutation flow-shop et standard flow-shop sont équivalents.*

Remarque : cette preuve ne fonctionne que parce qu'on est sur la première machine ou la dernière machine et que changer l'ordre sur celles-ci n'influe pas sur le reste. Elle ne fonctionne plus entre les machines 2 et 3 si la machine 3 n'est pas la dernière.

3.3.2 Complexité

Dans le cas du *permutation flow shop*, peu importe le nombre de machines, il y a a priori $n!$ ordres à tester pour trouver le meilleur, sauf à trouver un moyen simple d'en éliminer. Il convient de remarquer que c'est une croissance du nombre de cas à tester très rapide puisqu'avec 15 tâches, nous sommes déjà à à peu près 1300 milliards. Ceci est tout-à-fait accessible avec un ordinateur. Mais, pour 52 tâches, nous en sommes à 8×10^{67} (nombre de mélanges d'un paquet de 52 cartes). Il n'est donc pas question de tout tester.

Dans le cas du *standard flow shop*, comme on peut permuter entre les machines (sauf entre la première et la deuxième et l'avant-dernière et la dernière), le nombre de combinaisons à tester est $(n!)^{m-2}$. Avec 15 activités et 15 machines, cela représente déjà environ 3×10^{157}, beaucoup plus que d'atomes dans l'univers (nombre estimé à environ 10^{80}).

Mis à part dans le cas de deux machines (voir section **??**) où on connaît un algorithme fournissant la solution optimale en temps polynomial, ce problème est NP-difficile dès qu'il y a trois machines ou plus (moins difficile pour 3 que pour plus). La question à poser est une question de décision : existe-t-il une solution admissible finissant en moins de temps que T ? Vérifier qu'une solution admissible finit avant T est très rapide (linéaire en $m \times n$). Mais il n'existe pas d'algorithmes polynomiaux en n et m répondant par oui ou non à cette question (sauf si $P = NP$). C'est pourquoi il est intéressant pour les applications de trouver des algorithmes qui trouvent de « bonnes » solutions en temps raisonnable (voir section **??**).

3.3.3 Trouver des bornes sur le temps optimal

Il y a plusieurs bornes faciles à obtenir. Par exemple, on peut regarder la somme des temps d'exécution sur la dernière machine et y ajouter le minimum de la somme des temps d'exécution d'une activité sur toutes les machines sauf la dernière. En effet, le travail ne pourra pas commencer sur la dernière machine tant qu'une activité ne sera pas passée sur toutes les machines précédentes. Ainsi on a

$$T \geq \left(min_{i=1,\ldots,n} \sum_{j=1}^{m-1} T_{i,j} \right) + \sum_{i=1}^{n} T_{i,m} \, .$$

Mais ce résultat vaut aussi pour la première machine :

$$T \geq \left(min_{i=1,\ldots,n} \sum_{j=2}^{m} T_{i,j} \right) + \sum_{i=1}^{n} T_{i,1} \, .$$

On peut bien entendu également obtenir des bornes par rapport aux machines intermédiaires, un peu plus compliquées à écrire mais pas plus difficiles dans l'esprit.

3.3.4 Algorithmes naïfs

Cela ne veut pas dire qu'ils sont idiots, bien au contraire. Ils viennent à l'esprit assez rapidement et en voir l'efficacité et les limites est toujours utile. Nous n'allons les décrire que dans le cas $m = 2$, c'est-à-dire avec deux machines. Ces algorithmes sont adaptables pour $m \geq 3$ même s'ils sont parfois moins naturels et encore moins efficaces.

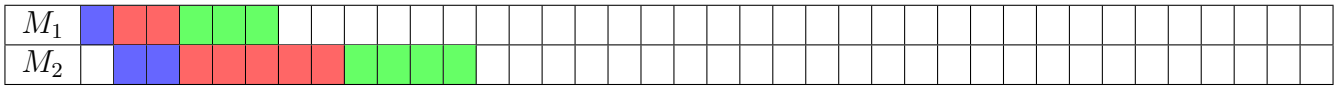
Les deux algorithmes sont basés sur les bornes obtenues ci-dessus. La première idée est de libérer la machine 2 le plus rapidement possible, c'est-à-dire de commencer par la tâche la plus courte sur la machine 1 afin de pouvoir travailler sur la machine 2 le plus tôt possible. Afin d'essayer de limiter les trous dans la machine 2, il est ensuite naturel de continuer à ordonner les tâches par temps d'exécution croissant sur la machine 1.

Algorithme 1 - Ordonner les tâches par ordre de temps d'exécution croissant sur la machine 1.

Voici un exemple où cet algorithme est optimal :

	M_1	M_2
A	2	5
B	1	2
C	3	4

En suivant l'algorithme ci-dessus, on trouve qu'il faut organiser les tâches dans l'ordre : B, A, C. On se retrouve alors avec

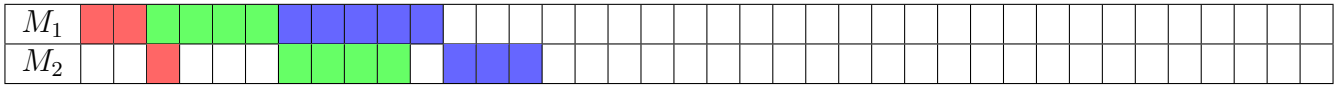


pour un temps total de 12 unités qui est clairement optimal car on ne peut pas commencer sur la 2ème machine avant le temps 2 et il faut 11 unités de temps sur la deuxième machine.

Mais cet algorithme n'est pas tout le temps optimal comme montré par l'exemple suivant :

	M_1	M_2
A	2	1
B	5	3
C	4	4

Si on suit la procédure ci-dessus, on trouve qu'il faut organiser les tâches dans l'ordre A, C, B, ce qui donne le résultat suivant :



pour un temps total de 14 unités. Il est possible de faire mieux (voir ci-dessous).

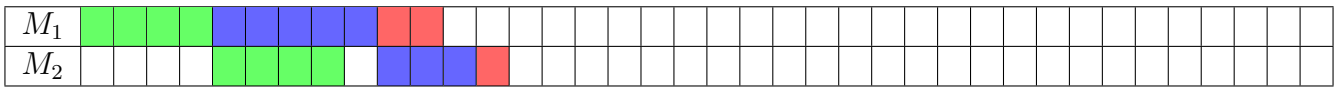
Une autre piste serait de regarder l'autre borne obtenue sur le temps optimal et d'essayer de terminer par la tâche la plus courte sur la deuxième machine. On peut alors imaginer ordonner les tâches dans l'ordre décroissant des temps d'exécution sur la deuxième machine.

Algorithme 2 - Ordonner les tâches par ordre de temps d'exécution décroissant sur la machine 2.

Si on reprend l'exemple ci-dessus :

	M_1	M_2
A	2	1
B	5	3
C	4	4

on trouve cette fois-ci qu'il faudrait ordonner selon C, B, A. Cela donne

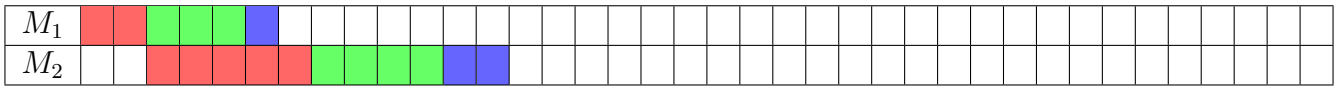


pour un temps total de 13 unités qui est, cette fois, optimal puisqu'il sature la deuxième borne (temps d'exécution des tâches sur la première machine + temps le plus court sur la deuxième).

Par contre, cet algorithme n'aurait pas fonctionné sur le premier exemple ci-dessus :

	M_1	M_2
A	2	5
B	1	2
C	3	4

Ce second algorithme aurait donné l'ordre A, C, B qui fournit la solution



pour un temps total de 13 unités alors que le premier algorithme fournissait le temps d'exécution optimal de 12 unités.

Est-ce que la solution optimale est toujours donnée soit par le premier algorithme soit par le second ? Aucune raison et on trouvera des contre-exemples dans la section ci-dessous.

3.4 L'algorithme de Johnson et son optimalité

Dans le cas $m = 2$ (2 machines), il existe un algorithme rapide qui trouve la solution optimale. Celui-ci combine les idées des deux algorithmes naïfs ci-dessus, commencer par les tâches les plus courtes sur la machine 1 et finir par les tâches les plus courtes sur la machine 2. Mais il résout optimalement les conflits potentiels entre ces deux algorithmes.

Pour commencer, afin de trouver cet algorithme, imaginons qu'on a un ordre de passage $1, 2, \ldots, n$ et regardons si on peut gagner du temps (ou au moins ne pas en perdre) en échangeant deux tâches i et $i+1$ pour un certain $1 \leq i \leq n-1$. Dans l'ordre initial, l'activité i commence au temps

$$t_{i,1} = \sum_{j=1}^{i-1} T_{j,1} = S_1$$

sur la première machine. On le note S_1 car ce temps ne dépend pas des tâches i et $i+1$ mais seulement de celles qui se trouvent avant. Et l'activité $i+1$ commence sur la première machine au temps

$$t_{i+1,1} = t_{i,1} + T_{i,1} = S_1 + T_{i,1} \, .$$

Sur la deuxième machine, l'activité i commence au temps

$$t_{i,2} = \max\{t_{i-1,2} + T_{i-1,2}, t_{i,1} + T_{i,1}\} = \max\{S_2, S_1 + T_{i,1}\}$$

et l'activité $i+1$ au temps

$$t_{i+1,2} = \max\{t_{i,2} + T_{i,2}, t_{i+1,1} + T_{i+1,1}\} = \max\{S_2 + T_{i,2}, S_1 + T_{i,1} + T_{i,2}, S_1 + T_{i,1} + T_{i+1,1}\} \, .$$

La machine 1 est donc libre pour les tâches suivantes au temps

$$S_1 + T_{i,1} + T_{i+1,1}$$

et la machine 2 au temps

$$\max\{S_2 + T_{i,2}, S_1 + T_{i,1} + T_{i,2}, S_1 + T_{i,1} + T_{i+1,1}\} + T_{i+1,2} \, .$$

Echanger les activités i et $i+1$ fait gagner du temps (ou au moins n'en fait pas perdre) si les deux machines sont libérées avant ou au même moment en cas d'échange, c'est-à-dire si

$$S_1 + T_{i+1,1} + T_{i,1} \leq S_1 + T_{i,1} + T_{i+1,1} \ ,$$

ce qui est toujours le cas, et si

$$\begin{aligned} &\max\left\{S_2 + T_{i+1,2}, S_1 + T_{i+1,1} + T_{i+1,2}, S_1 + T_{i+1,1} + T_{i,1}\right\} + T_{i,2} \\ &\quad \leq \max\left\{S_2 + T_{i,2}, S_1 + T_{i,1} + T_{i,2}, S_1 + T_{i,1} + T_{i+1,1}\right\} + T_{i+1,2} \ , \end{aligned}$$

ce qui est moins clair. Nous pouvons réécrire cette condition en une conjonction de trois conditions :

$$\begin{aligned} &S_2 + T_{i+1,2} + T_{i,2} \\ &\leq \max\left\{S_2 + T_{i,2} + T_{i+1,2}, S_1 + T_{i,1} + T_{i,2}, +T_{i+1,2}, S_1 + T_{i,1} + T_{i+1,1} + T_{i+1,2}\right\} \ , \end{aligned}$$

ce qui est toujours vérifié, et

$$\begin{aligned} &S_1 + T_{i+1,1} + T_{i+1,2} + T_{i,2} \\ &\leq \max\left\{S_2 + T_{i,2} + T_{i+1,2}, S_1 + T_{i,1} + T_{i,2} + T_{i+1,2}, S_1 + T_{i,1} + T_{i+1,1} + T_{i+1,2}\right\} \end{aligned} \tag{3.3}$$

et

$$\begin{aligned} &S_1 + T_{i+1,1} + T_{i,1} + T_{i,2} \\ &\leq \max\left\{S_2 + T_{i,2} + T_{i+1,2}, S_1 + T_{i,1} + T_{i,2} + T_{i+1,2}, S_1 + T_{i,1} + T_{i+1,1} + T_{i+1,2}\right\} \ . \end{aligned} \tag{3.4}$$

Il suffit donc que

$$T_{i+1,1} \leq T_{i,1} \text{ ou } T_{i,2} \leq T_{i,1}$$

et

$$T_{i+1,1} \leq T_{i+1,2} \text{ ou } T_{i,2} \leq T_{i+1,2}$$

pour pouvoir faire l'échange sans perte de temps. Cela peut également s'écrire

$$\min\left\{T_{i+1,1}, T_{i,2}\right\} \leq T_{i,1} \text{ et } \min\left\{T_{i+1,1}, T_{i,2}\right\} \leq T_{i+1,2}$$

ou encore

$$\min\left\{T_{i+1,1}, T_{i,2}\right\} \leq \min\left\{T_{i,1}, T_{i+1,2}\right\} \ . \tag{3.5}$$

Ainsi, si on part d'un ordre de passage, tant qu'il y a un indice i pour lequel (**??**) est vérifié, on peut échanger les tâches i et $i+1$ sans perdre de temps au total. Cela démontre le résultat suivant :

Théorème 3.2. *Pour $m = 2$, l'ordre optimal peut être cherché dans l'ensemble des ordres de passage satisfaisant*

$$\min\left\{T_{i+1,1}, T_{i,2}\right\} \geq \min\left\{T_{i,1}, T_{i+1,2}\right\} \ \textit{pour tout } 1 \leq i \leq n-1 \ .$$

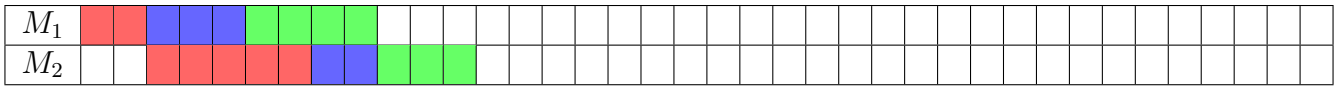
Cela ne signifie pas que tout ordre optimal vérifie cette condition mais que l'ordre optimal, qui existe puisque c'est juste celui correspondant au temps total d'exécution minimal parmi un nombre fini de possibilités, peut être remplacé par un autre vérifiant cette condition si jamais il ne la vérifie pas. Il suffit donc d'aller chercher parmi les ordres de passage vérifiant cette condition. Un exemple d'ordre optimal ne satisfaisant pas la condition est donné par

	M_1	M_2
A	2	5
B	3	2
C	4	3

dans cet ordre qui ne vérifie pas

$$2 = \min\{T_{3,1}, T_{2,2}\} \geq \min\{T_{2,1}, T_{3,2}\} = 3$$

et qui donne la solution



qui est optimale puisqu'elle atteint la borne inférieure donnée par le temps minimal d'exécution d'une tâche sur la première machine plus le temps total d'exécution sur la deuxième machine. Bien sûr, on vérifiera que l'ordre ACB vérifie lui la condition du théorème **??** et est tout aussi optimal.

Mais, plus embêtant, il va falloir creuser un peu plus car tous les ordres de passage vérifiant cette condition ne sont pas optimaux. Prenons l'exemple suivant :

	M_1	M_2
A	3	2
B	1	1
C	2	3

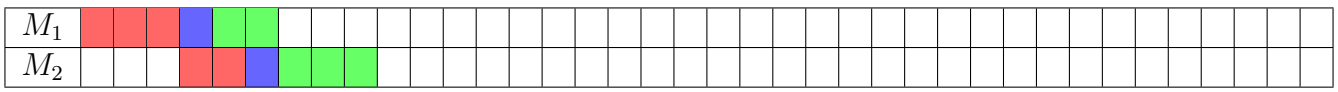
dans cet ordre qui vérifie bien

$$1 = \min\{T_{2,1}, T_{1,2}\} \geq \min\{T_{1,1}, T_{2,2}\} = 1$$

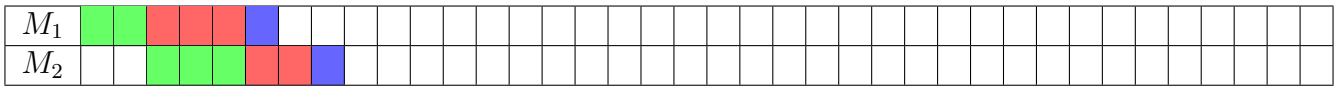
et

$$1 = \min\{T_{3,1}, T_{2,2}\} \geq \min\{T_{2,1}, T_{3,2}\} = 1$$

et qui donne la solution



en 9 unités de temps alors qu'il existe une solution meilleure



en 8 unités de temps.

Le problème semble venir de la tâche B qui a deux temps d'exécution égaux, qui plus est minimaux. Ainsi, en la mettant au milieu, on est certain de vérifier la condition du théorème alors que, bien sûr, l'heuristique des algorithmes naïfs nous recommanderait de commencer par elle ou de finir par elle.

Afin de casser ce problème, on va supposer que toutes nos activités ont des temps d'exécution différents, on suppose donc que

$$T_{i,\alpha} \neq T_{j,\beta} \text{ pour tous } 1 \leq i, j \leq n \text{ et } 1 \leq \alpha, \beta \leq 2 \text{ avec } (i,\alpha) \neq (j,\beta)\ . \tag{3.6}$$

Remarquons que le temps total d'exécution, dans un ordre donné, est continu dans les temps d'exécution des activités donc de petites perturbations des temps d'exécution des activités permettent de satisfaire la condition ci-dessus sans changer beaucoup le temps total d'exécution. En passant à la limite, on pourra même montrer qu'un ordre optimal pour des perturbations reste optimal à la limite.

Théorème 3.3. *Si $m = 2$ et si les temps d'exécution des n activités sur les deux machines vérifient* (**??**)*, alors il existe un unique ordre de passage vérifiant la condition du théorème* **??** *(qu'on supposera être $1, 2, \ldots, n$). Celui-ci vérifie de plus qu'il existe $1 \leq i_0 \leq n$ tel que :*

- *pour $1 \leq i \leq i_0$, on a $T_{i,1} < T_{i,2}$ et pour $i_0 + 1 \leq i \leq n$, on a $T_{i,1} > T_{i,2}$.*
- *pour $1 \leq i \leq j \leq i_0$, on a $T_{i,1} < T_{j,1}$.*
- *pour $i_0 \leq i \leq j \leq n$, on a $T_{i,2} > T_{j,2}$.*

Preuve - Soit $1, 2, \ldots, n$ un ordre de passage vérifiant (**??**) et la condition du théorème **??**. Soit $1 \leq i \leq n-1$, supposons que $T_{i,1} > T_{i,2}$. Alors on a nécessairement $T_{i+1,1} > T_{i+1,2}$ et $T_{i+1,2} < T_{i,2}$. En effet, la condition du théorème **??** donne

$$\min\{T_{i+1,1}, T_{i,2}\} \geq \min\{T_{i,1}, T_{i+1,2}\}$$

ce qui impose avec $T_{i,1} > T_{i,2}$ que

$$T_{i+1,2} < T_{i,2} < T_{i,1} \text{ et } T_{i+1,2} < T_{i+1,1} .$$

On a donc montré que

$$\text{pour } 1 \leq i \leq n-1,\ T_{i,1} > T_{i,2} \Longrightarrow T_{i+1,1} > T_{i+1,2} \text{ et } T_{i+1,2} < T_{i,2} .$$

Exactement de la même façon, on montre que

$$\text{pour } 1 \leq i \leq n-1,\ T_{i+1,1} < T_{i+1,2} \Longrightarrow T_{i,1} < T_{i,2} \text{ et } T_{i+1,1} > T_{i,1} .$$

Ces deux implications permettent de démontrer le théorème. L'unicité, et d'ailleurs toute la preuve, repose sur le fait que les temps d'exécution ne sont jamais égaux. ♢

Par approximation, on a le corollaire suivant :

Corollaire 3.2 (algorithme de Johnson)**.** *Pour $m = 2$, une solution donnant le temps optimal d'exécution est donné par les conditions suivantes : le groupe $G_1 = \{1, \ldots, i_0\}$ contient des activités satisfaisant $T_{i,1} \leq T_{i,2}$ rangées par ordre croissant de temps d'exécution sur la première machine ; le groupe $G_2 = \{i_0 + 1, \ldots, n\}$ contient des activités satisfaisant $T_{i,1} \geq T_{i,2}$ rangées par ordre décroissant de temps d'exécution sur la deuxième machine. L'ordre final est donné par les activités du groupe G_1 dans l'ordre suivi de celles du groupe G_2 dans l'ordre.*

Cet ordre de passage n'est pas nécessairement le seul optimal, ce n'est pas non plus nécessairement le seul à vérifier les conditions du théorème **??** s'il y a des égalités de temps d'exécution. Mais tous ces ordres donnent le même temps d'exécution total. On remarquera en particulier qu'une tâche vérifiant $T_{i,1} = T_{i,2}$ peut être mise indifféremment dans le groupe G_1 ou dans le groupe G_2, ce qui changera drastiquement sa position dans l'ordre de passage obtenu.

On peut donc écrire l'algorithme de Johnson, qui donne toujours une solution optimale au problème de flow-shop sur 2 machines :

Algorithme de Johnson (1954)

- Pour $i = 1$ à n, si $T_{i,1} \leq T_{i,2}$, mettre l'activité i dans le groupe G_1. Sinon, la mettre dans le groupe G_2.
- Trier les éléments de G_1 par ordre croissant des $T_{i,1}$.
- Trier les éléments de G_2 par ordre décroissant des $T_{i,2}$.
- Un ordre optimal est donné par les éléments de G_1 dans l'ordre obtenu suivis des éléments de G_2 dans l'ordre obtenu.

Cet algorithme dépendant essentiellement d'algorithmes de tri, il a une complexité en $O(n \log n)$. A titre de remarque, on a fait le choix de mettre les activités pour lesquelles $T_{i,1} = T_{i,2}$ dans le groupe G_1 mais on pourrait les mettre dans le groupe G_2 voire alterner.

Une autre version de cet algorithme consiste à faire les choses suivantes : on trie tous les temps par ordre croissant, qu'ils soient sur la première ou la deuxième machine. On prend le plus petit temps qui correspond à une activité i. Si ce temps est sur la première machine, on place cette activité en tête de liste. Si ce temps est sur la deuxième machine, on place cette activité en queue de liste. On élimine l'activité et on prend le temps suivant qui correspond à une activité j. Si le temps correspondant est sur la première machine, on place cette activité en suite de début de liste ; sinon, on la place juste avant la queue de liste. On balaie ainsi toutes les activités. Le critère du corollaire est trivialement vérifié puisqu'il est vérifié à chaque insertion.

Faisons fonctionner l'algorithme de Johnson sur un exemple : voici les activités et leurs temps d'exécution sur les deux machines.

	M_1	M_2
A	5	6
B	7	3
C	4	4
D	3	5
E	2	1
F	8	5
G	4	2
H	4	5

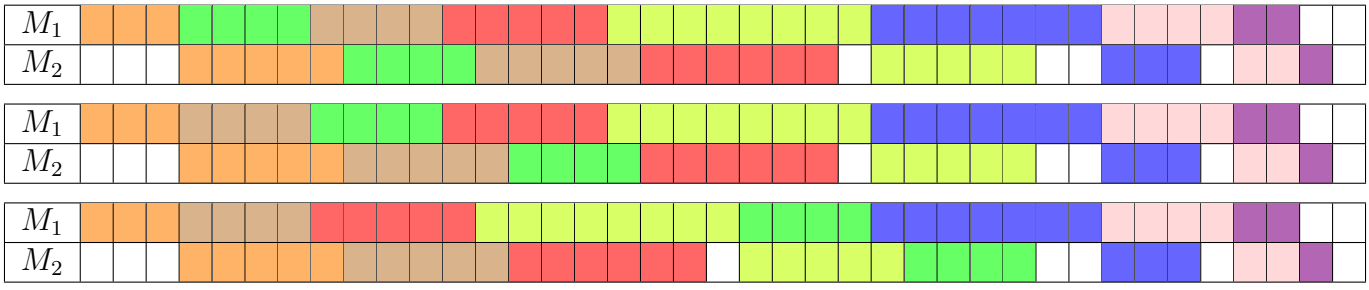
Vont dans le groupe G_1 les activités A, C, D, H et dans le groupe G_2 les activités B, E, F, G. Si on trie celles de G_1 par ordre croissant de temps d'exécution sur la première machine, on obtient $DCHA$ ou $DHCA$. Si on trie celles de G_2 par ordre décroissant de temps d'exécution sur la deuxième machine, on obtient $FBGE$. Cela donne déjà deux ordres possibles :

- DCHAFBGE
- DHCAFBGE

Mais l'algorithme de Johnson ne stipule pas dans quel groupe mettre C et nous aurions pu la mettre dans le groupe G_2, ce qui donne un autre ordre possible :

- DHAFCBGE

Voici les trois solutions que cela donne :



3.5 Heuristiques avec plus de machines

Dès que $m \geq 3$, il n'existe pas d'algorithme donnant la solution optimale à coup sûr en temps polynomial. On cherche donc des heuristiques qui vont donner des bonnes solutions en temps raisonnable. On en présente deux dans la suite même si, pour $m = 3$ en particulier, il en existe d'autres très efficaces [7].

On ne considère ici que le problème du *permutation flow-shop*.

3.5.1 Heuristique de Campbell, Dudek et Smith (CDS, 1970)

Elle consiste à utiliser l'algorithme de Johnson en regroupant des machines. Par exemple, avec trois machines, on pourra regrouper les machines 1 et 2 et faire un algo de Johnson entre 1-2 et 3. Avec 4 machines, on peut faire 1-2 et 3-4. On peut tester tous ces regroupements et garder le meilleur. C'est en temps pas plus que polynomial et donne des résultats satisfaisants.

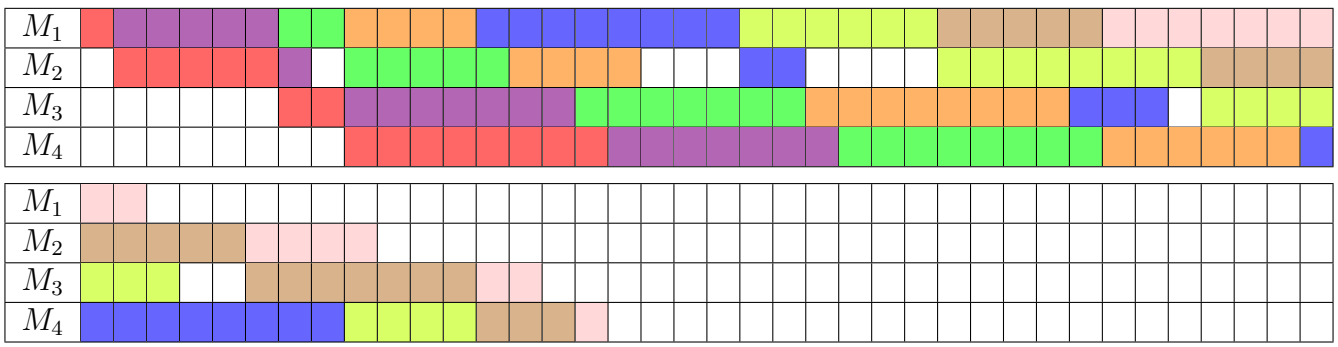
Regardons cette heuristique sur un exemple :

	M_1	M_2	M_3	M_4
A	1	5	2	8
B	8	2	3	9
C	2	5	7	8
D	4	4	8	6
E	5	1	7	7
F	6	8	7	4
G	9	4	2	1
H	5	9	7	3

On va regrouper les machines 1 et 2 et les machines 3 et 4. Ceci donne :

	$M_1 - M_2$	$M_3 - M_4$
A	6	10
B	10	12
C	7	15
D	8	14
E	6	14
F	14	11
G	13	3
H	14	10

et on applique l'algorithme de Johnson pour ces deux nouvelles machines. Dans le groupe G_1, il y a A, B, C, D, E et dans le groupe G_2, il y a F, G, H. Le tri donne alors l'ordre AECDBFHG, sachant que A et E peuvent être permutés (et ça vaut le coup d'essayer les deux ici qui peuvent ne pas donner le même résultat). On trouve alors



7. Références.

et il se trouve que c'est un ordre de passage optimal. En effet, il est impossible de commencer sur la machine 4 avant le temps 8 car le minimum de la somme des temps d'exécution sur les trois premières machines parmi toutes les activités est pour l'activité A avec un total de 8. C'est bien elle qui commence ici et il n'y a aucun trou sur la machine 4. Cet ordonnancement est donc optimal.

Mais ce n'est pas toujours le cas.

3.5.2 Heuristique de Nawaz, Enscore et Ham (NEH), 1983

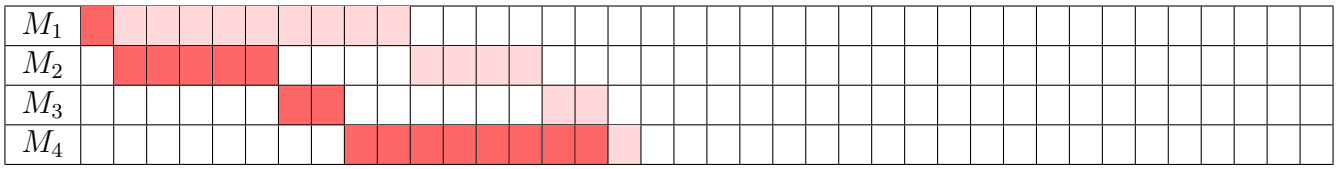
Elle consiste à classer les tâches par temps total (somme totale sur toutes les machines) dans l'ordre croissant. On classe ensuite les deux premières au mieux puis on les insère successivement en testant toutes les possibilités pour voir la meilleure. Pas très difficile à mettre en oeuvre et donne des résultats pas mauvais.

Regardons cette heuristique sur le même exemple :

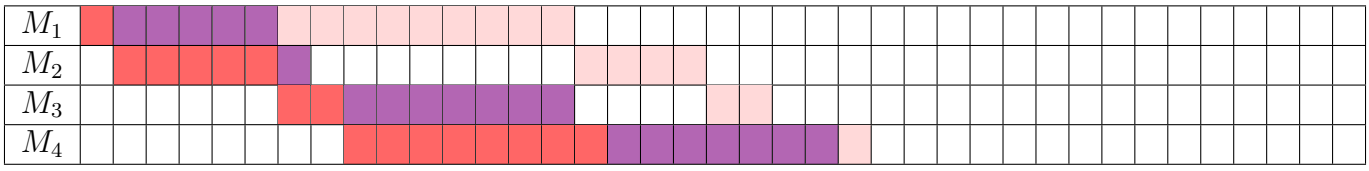
	M_1	M_2	M_3	M_4
A	1	5	2	8
B	8	2	3	9
C	2	5	7	8
D	4	4	8	6
E	5	1	7	7
F	6	8	7	4
G	9	4	2	1
H	5	9	7	3

Les temps totaux donnent l'ordre suivant : A (16), G(16), E(20), B(22), C(22), D(22), H(24), F(25). Encore une fois, il y aurait des choix à faire et à tester pour l'ordre dans lequel traiter les trois activités B, C et D. Et il y a aussi des choix dans les insertions qui ne sont pas nécessairement uniques.

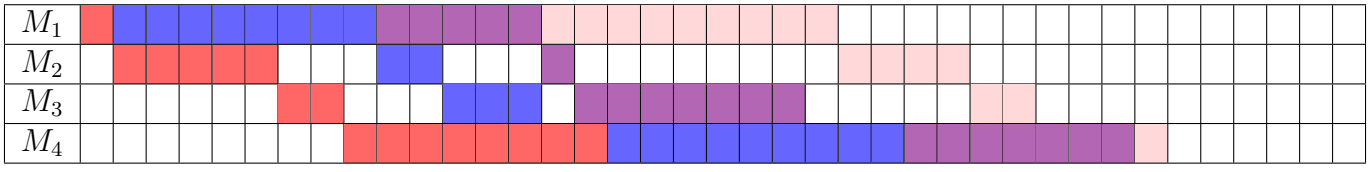
En regardant les deux premières, A et G, il n'est pas difficile de voir qu'il vaut mieux les mettre dans cet ordre-là :



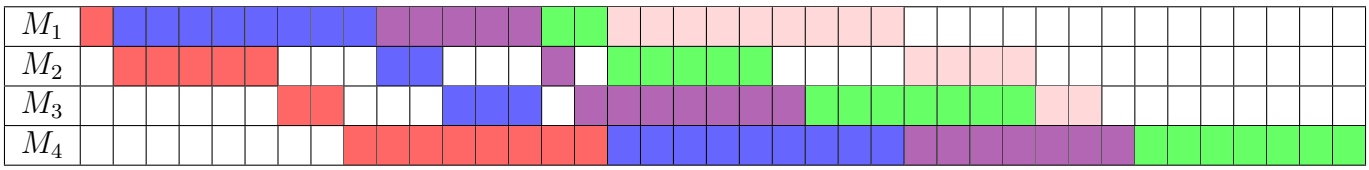
Essayons maintenant d'insérer E au mieux. On ne peut pas faire mieux que :

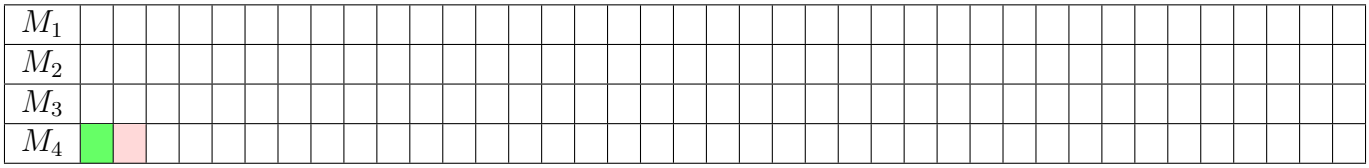


Essayons maintenant d'insérer B au mieux. On ne peut pas faire mieux que :

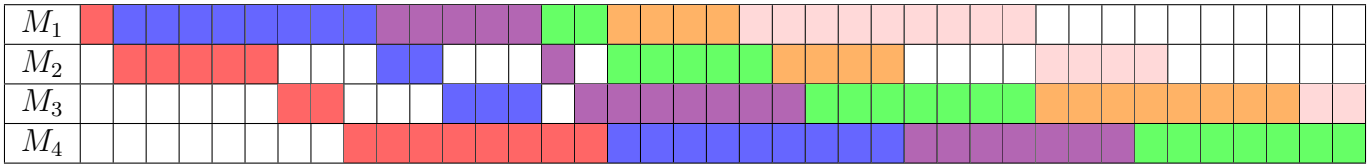


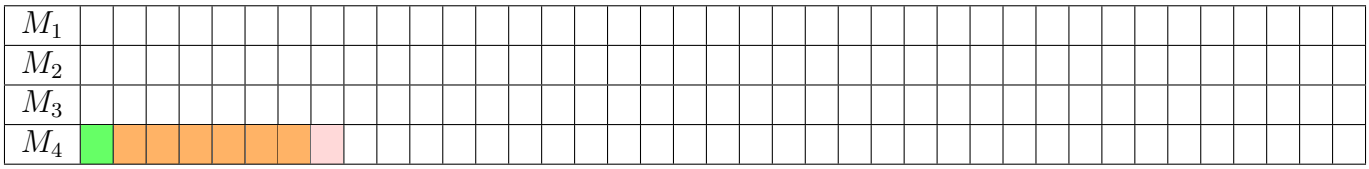
Essayons maintenant d'insérer C au mieux. On ne peut pas faire mieux que :



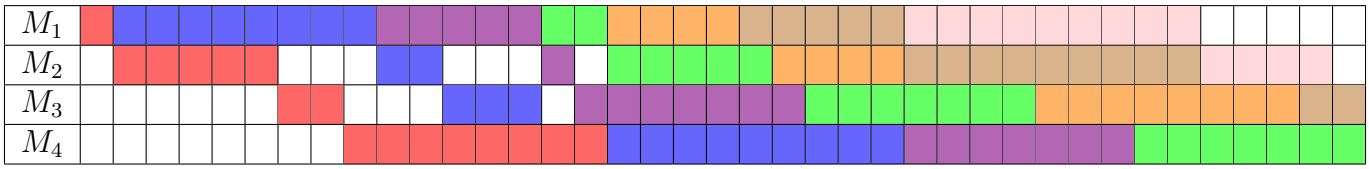


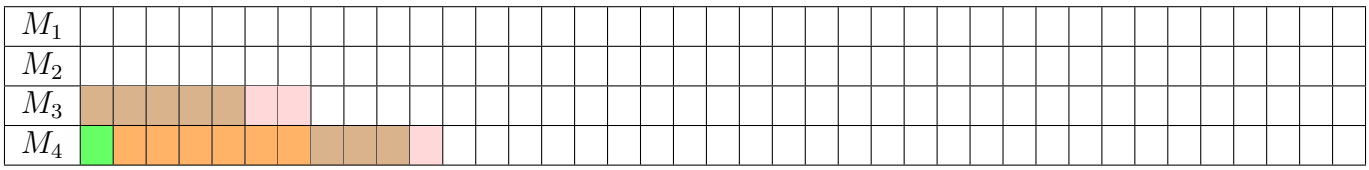
Essayons maintenant d'insérer D au mieux. On ne peut pas faire mieux que :



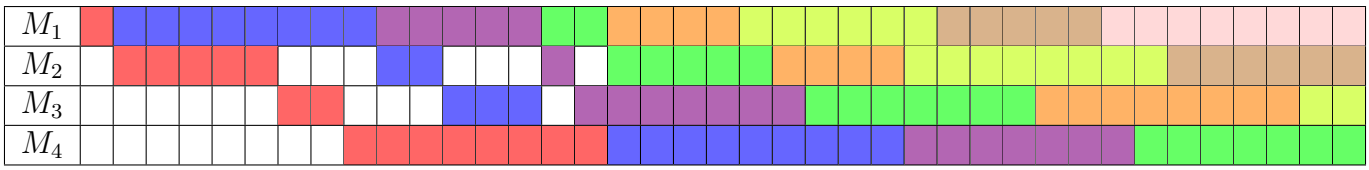


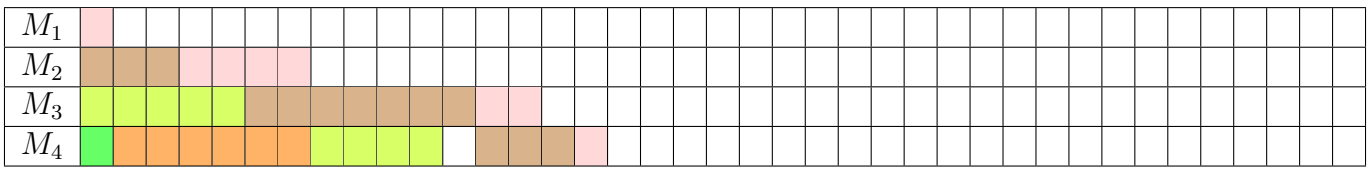
Essayons maintenant d'insérer H au mieux. On ne peut pas faire mieux que :





Essayons enfin d'insérer F au mieux. On ne peut pas faire mieux que :





C'est très bon mais ce n'est pas optimal (voir sous-section précédente) à une unité près.

3.6 Quelques exemples avec 4 machines

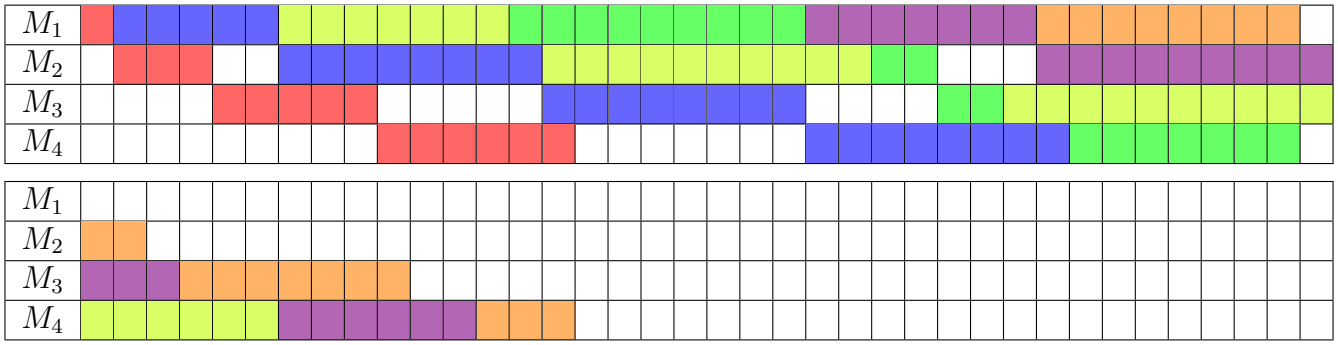
L'objectif est de donner quelques exemples où la solution du *standard flow-shop* et celle du *permutation flow-shop* ne sont pas les mêmes. En passant, on donnera aussi des exemples où l'heuristique CDS ne donne pas le résultat optimal dans le *permutation flow-shop*.

3.6.1 Exemple 1

Voici les activités et leurs temps d'exécution sur les quatre machines.

	M_1	M_2	M_3	M_4
A	1	3	5	6
B	5	8	8	8
C	9	2	2	7
D	8	2	7	3
E	7	9	3	6
F	7	10	10	6

Voici la solution optimale et elle est unique :



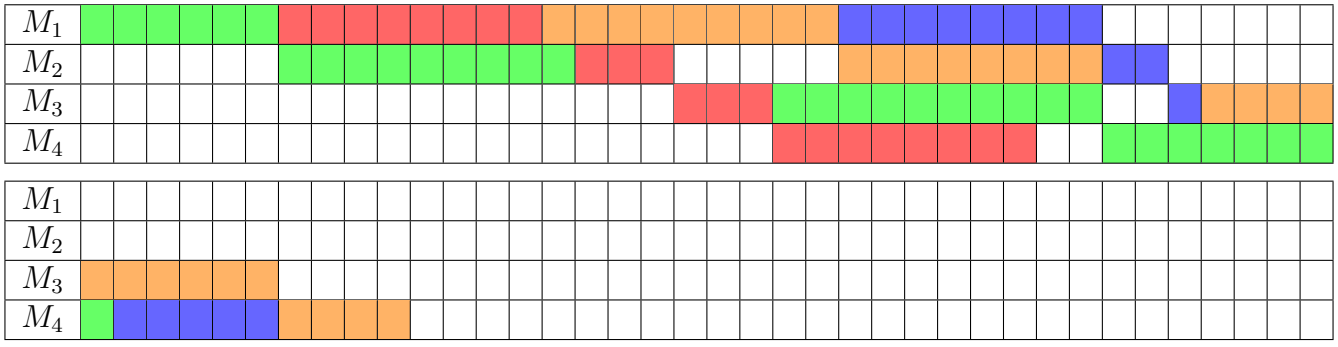
pour un total de 53. Le temps optimal sans permutation de l'ordre entre deux machines est de 56. Il y a alors 4 ordres optimaux dont aucun n'aurait été donné par l'heuristique CDS utilisée comme ci-dessus puisqu'il s'agit des ordres ABFCDE, ABFCED, ABFECD, ABFEDC.

3.6.2 Exemple 2

Voici les activités et leurs temps d'exécution sur les quatre machines.

	M_1	M_2	M_3	M_4
A	8	3	3	8
B	8	2	1	5
C	6	9	10	8
D	9	8	10	4

Voici la solution optimale et elle est unique :



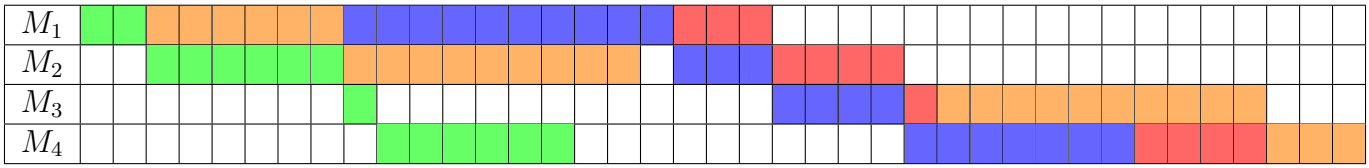
pour un temps total de 48. A noter la double permutation des deux premières tâches et des deux dernières entre les machines 2 et 3. L'ordre optimal sans autoriser de permutation d'ordre est CADB pour un temps de 50.

3.6.3 Exemple 3

Voici les activités et leurs temps d'exécution sur les quatre machines.

	M_1	M_2	M_3	M_4
A	3	4	1	4
B	10	3	4	7
C	2	6	1	6
D	6	9	10	3

Voici la solution optimale et elle est unique :



pour un temps total de 39. C'est une permutation sur trois activités entre les machines 1 et 2. Le temps optimal sans changement d'ordre est de 41 et il est donné par l'ordre CDAB qui n'apparaît sur aucune machine ci-dessus.

3.7 Bibliographie

- Recherche opérationnelle, *wikipedia*.
- Théorie de l'ordonnancement, *wikipedia*.
- Diagramme de Gantt, *wikipedia*.
- Problème de gestion de projet à contraintes de ressources, *wikipedia*.
- PERT, *wikipedia*.
- *Gestion de projet - réaliser le diagramme de Pert*, G. Casanova et D. Abécassis.
- *Modèles et algorithmes en ordonnancement*, Groupe Gotha, Ellipses, 2004.
- Algorithme de Bellman Ford, *wikipedia*.
- Ordonnancement dans les systèmes d'exploitation, *wikipedia*.
- Microprocesseur multi-cœur, *wikipedia*.
- Ordonnancement d'atelier, *wikipedia*.
- Flow-shop, *wikipedia*.

Chapitre 4

Théorie de la chaleur

4.1 Les différents modes de cuisson

4.2 La théorie de la conduction de la chaleur

4.3 Le cas statique - problème de Dirichlet

4.3.1 Le cas discret

4.3.2 Le cas continu

Chapitre 5

Mélanges et entropie

Chapitre 6

Empilement optimal

6.1 Pavages dans le plan

6.2 Le cas de domaines bornés

6.3 Dans l'espace

6.3.1 La conjecture de Kepler

6.4 En plus grandes dimensions

Chapitre 7

Espionnage

7.1 Les différentes techniques d'espionnage

7.2 Les attaques par canaux auxiliaires

Chapitre 8

Probabilités et statistiques

8.1 La loi des grands nombres

8.2 Proba théorique vs statistiques

Chapitre 9

Jeu à deux joueur·ses

9.1 Définition générale

9.2 Stratégie gagnante

9.3 Exemples de jeux à deux joueur·ses

9.4 Autres jeux et stratégies

Chapitre 10

Théorème de Borsuk-Ulam

10.1 Qu'est-ce que la topologie ?

10.2 Le théorème des valeurs intermédiaires

10.2.1 Le cas continu

10.2.2 Le cas discret

10.3 Le théorème de Borsuk-Ulam

10.4 Applications

Chapitre 11

Optimisation linéaire

Chapitre 12

Percolation

Chapitre 13

Problèmes de partage

Chapitre 14

Mousses et bulles de savon

Chapitre 15

Bibliographie

Une petite bibliothèque consacrée à l'IA est disponible à la MMI avec un ensemble d'ouvrages à consulter sur place, notés par  . Merci d'en prendre soin :-)

15.1 Des ouvrages, films, etc. à conseiller au public

Voici quelques ouvrages, textes de vulgarisation accessibles à tout public.

15.2 Des ouvrages, sites, références pour en savoir plus