École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
Fax +33 (0)2 47 36 14 22
polytech.univ-tours.fr

Département Informatique
5e année
2014-2015

Rapport de Projet de Fin d'Études

Algorithmes exponentiels en ordonnancement

Encadrants
Lei Shang
lei.shang@univ-tours.fr
Vincent T'Kindt
vincent.tkindt@univ-tours.fr
Christophe Lenté
christophe.lente@univ-tours.fr

École Polytechnique
Université François Rabelais
Tours

Étudiant
Pierre-Antoine Morin
pierre-antoine.morin@etu.univ-tours.fr

DI5 2014-2015

Table des matières

Table des figures

Liste des tableaux

Introduction

Dans le cadre du parcours en cycle Ingénieur Informatique à Polytech Tours, les élèves de cinquième et dernière année ont l'opportunité de travailler pendant une année entière sur un Projet de Fin d'Études, dont le but est de mettre en œuvre les compétences qu'ils ont pu acquérir tout au long de leur formation.

Ce Projet de Fin d'Études, intitulé « Algorithmes exponentiels en ordonnancement », s'inscrit dans le travail de thèse de M. Lei Shang, doctorant à Polytech Tours. Ce projet est co-encadré par M. Vincent T'Kindt et M. Christophe Lenté, enseignants-chercheurs dans cette même École.

Ce projet est très fortement typé « Recherche Opérationnelle ». Il porte sur l'analyse d'algorithmes exacts, permettant de résoudre des problématiques d'ordonnancement, discipline dédiée à l'étude de l'organisation de systèmes définis par des tâches soumises à des contraintes de types variés, par exemple en termes de dates ou de ressources.

Dans ce rapport, la gestion de projet est le premier point évoqué. Ensuite, un état de l'art est proposé, pour définir ce qu'est un algorithme exact, et montrer comment ce type d'algorithme peut être utilisé dans le domaine de l'ordonnancement. Enfin, une étude de plusieurs algorithmes exacts résolvant le problème $F_3||C_{max}$ est menée, afin de dresser un comparatif des performances de ces méthodes.

Gestion de projet

Dans la mesure où la dimension « Recherche Opérationnelle » de ce PFE est prédominante, il est assez difficile de faire le parallèle avec les notions classiques abordées en gestion de projet. Dans cette partie, nous essaierons néanmoins de fournir le plus d'informations possibles à propos du déroulement et de la gestion du projet.

2.1 Découpage du projet en phases

Le projet a été séparé en deux phases bien distinctes.

1. La première a consisté à réaliser un état de l'art portant sur la mise en œuvre d'algorithmes exponentiels, dans le cadre des problématiques d'ordonnancement. Les objectifs de cette première phase étaient multiples : découvrir différentes familles « classiques » d'algorithmes exacts, comprendre et savoir estimer leur complexité temporelle et spatiale au pire cas, étudier des problèmes très variés pour élargir le champ culturel. Cette phase, très théorique, constitue un pré-requis indispensable pour la phase suivante.
2. La deuxième a porté sur l'étude du problème $F_3||C_{max}$ à travers des algorithmes exacts de divers types. Cette phase a été sous-découpée en plusieurs « cycles de production » : chaque cycle correspond à l'étude d'un algorithme en particulier, et comprend les étapes d'analyse ou de conception, d'implémentation et de test des performances.

2.2 Phase préliminaire : état de l'art

Pour mener à bien cette phase, plusieurs articles scientifiques ont été mis à la disposition de l'étudiant (cf. bibliographie à la fin du rapport, page 50). Chaque article aborde une ou plusieurs thématiques. Certaines ont été étudiées pour l'intérêt culturel qu'elles représentent ; d'autres, en plus de cela, ont apporté un savoir qui a pu être réinvesti ensuite, lors de l'implémentation.

2.3 Succession de cycles de production

Au total, quatre cycles de production ont eu lieu au cours de ce projet, c'est-à-dire que quatre algorithmes distincts ont pu être étudiés.

1. Un algorithme de programmation dynamique, dont la taille des résultats obtenus peut être réduite en utilisant une condition de dominance qui définit un front de Pareto
2. Un algorithme à base de « triplets » (qui représentent un chemin critique)
3. Un algorithme mêlant l'utilisation des « triplets » du deuxième algorithme et la condition de dominance du premier algorithme
4. Un algorithme de type *Branch & Bound*

Une proposition de variante pour le premier algorithme, conçue par l'étudiant, a également été proposée.

2.4 À propos du développement

Le langage de programmation choisi pour l'implémentation des algorithmes est le C++. D'une part, l'utilisation d'un langage compilé permet d'optimiser les performances pour l'exécution des programmes ; d'autre part, des outils évolués (par exemple : les conteneurs fournis par la STL ou par d'autres librairies) sont disponibles, permettant de faciliter le développement.

Les logiciels suivants ont été utilisés au cours du développement :

Visual Studio (version 2012) pour le codage, la compilation et le débogage

Valgrind (version 3.10.0) pour le traçage des fuites mémoire

Doxygen (version 1.8.6) pour la documentation du code

Remarque Le code produit a été intégralement documenté. Les commentaires contiennent non seulement des explications à propos des interactions entre les classes développées ainsi que le fonctionnement des fonctions et méthodes, mais aussi des démonstrations pas à pas de la complexité de certains traitements parfois complexes, qui constituent le noyau central des algorithmes implémentés.

2.5 Planning

La phase préliminaire (l'état de l'art) a duré pratiquement tout le premier semestre.

Le premier cycle de production (portant sur l'algorithme de programmation dynamique) a duré jusqu'à début février, et s'est terminé avec la proposition de la variante.

Les cycles de production suivants ont eu tendance à se chevaucher. La priorité a été donnée à l'algorithme à base de triplets (version initiale), suivi de l'intégration de la condition de dominance, en parallèle avec l'étude de l'algorithme *Branch & Bound*.

Remarque Les tâches à réaliser ont été définies au fur et à mesure du projet, en fonction de l'avancée du travail de l'étudiant. Par exemple, le nombre d'algorithmes à étudier n'était pas fixé dès le début du projet. Dans ses conditions, il était difficile de faire de la planification.

État de l'art

3.1 Qu'est-ce qu'un algorithme exponentiel ?

3.1.1 Définition

Un algorithme exponentiel est un algorithme dont la complexité temporelle et/ou spatiale au pire cas est exponentielle. Généralement, ce type d'algorithme correspond à une méthode de résolution exacte : les solutions trouvées sont optimales. Cependant, un coût en terme de durée de calcul nécessaire et/ou d'espace mémoire requis pour aboutir au résultat est engendré. Ce coût augmente exponentiellement avec la grandeur caractéristique des entrées de l'algorithme.

Remarque C'est la raison pour laquelle les algorithmes associés à des méthodes approchées ne doivent pas être exponentiels : le but n'est pas de trouver une solution optimale, mais une solution satisfaisante, en un temps de calcul rapide, pour une occupation mémoire réduite.

3.1.2 Notation $\mathcal{O}^*(\alpha^n)$

Étant donné un algorithme exponentiel, dont la « taille » des entrées est représentée par la grandeur caractéristique n, on dira que « la complexité de l'algorithme est en $\mathcal{O}^*(\alpha^n)$ » avec $\alpha \in \mathbb{R}^{+*}$, s'il existe un polynôme P tel que la complexité de cet algorithme est en $\mathcal{O}(P(n) \times \alpha^n)$.

3.2 Quelques grandes familles d'algorithmes exponentiels

3.2.1 *Branch & Bound*

Principe

Dans un algorithme de type *Branch & Bound (B&B)*, aussi appelé *Procédure par Séparation et Évaluation (PSE)* en français, l'espace des solutions est scindé en plusieurs partitions, selon une méthode récursive à définir en fonction du problème. Ce partitionnement peut être représenté sous la forme d'un arbre (*Branch*) où les nœuds sont des partitions :

- la racine est la partition qui contient toutes les solutions ;
- les nœuds intermédiaires sont des partitions dont la taille diminue avec la profondeur de l'arbre ;
- les feuilles sont des partitions contenant un seul élément, elles sont donc assimilables à des solutions.

À chaque nœud est associée une borne (*Bound*) à définir en fonction du critère étudié pour comparer les solutions. Cette borne est calculée au niveau du nœud, indépendamment des valeurs du critère, pour les solutions appartenant à ce nœud. Le calcul de cette borne doit être aussi rapide que possible (au pire cas, en temps polynomial). Les bornes vont permettre de ne pas parcourir l'arbre entièrement.

Dans le cas de la minimisation *(respectivement de la maximisation)* du critère, cette borne est appelée borne inférieure *(respectivement supérieure)* : elle doit être inférieure *(respectivement supérieure)* à toutes les valeurs de critère pour les solutions appartenant au nœud, et aussi grande *(respectivement petite)* que possible.

Dès lors, si la valeur du critère pour la meilleure solution connue à un instant donné est inférieure *(respectivement supérieure)* à la valeur de la borne inférieure *(respectivement supérieure)* d'un nœud, alors ce nœud peut être ignoré : il ne peut pas contenir de solution(s) optimale(s).

Complexité

La complexité d'un algorithme de type *Branch & Bound* dépend du partitionnement récursif et de la manière dont les bornes sont calculées au niveau de chaque nœud. Le pire cas est atteint lorsque ces bornes ne sont jamais utilisables pour couper des branches : l'algorithme devient alors une énumération totale (calcul de la valeur du critère pour toutes les solutions possibles), à laquelle viennent s'ajouter des calculs inutiles sur tous les nœuds. Autrement dit, la complexité au pire cas d'un algorithme de type *Branch & Bound* est très mauvaise, puisqu'elle est en $\mathcal{O}^*(n!)$, c'est-à-dire équivalente à celle d'une recherche brute force.

3.2.2 Programmation dynamique

Principe

Le terme « programmation dynamique » a été introduit par Richard Bellman dans les années 1940. La programmation dynamique est applicable aux problèmes dont la fonction objectif est la somme de fonctions monotones croissantes des ressources. Elle s'appuie sur le théorème suivant :

Théorème (Optimalité)**.** *Toute politique optimale est constituée de sous-politiques optimales.*

La stratégie d'un algorithme de programmation dynamique est de déduire une solution optimale pour un problème $\mathcal{P}$ initial à partir des solutions aux sous-problèmes qui forment le problème $\mathcal{P}$.

Pour concevoir un algorithme de programmation dynamique, il faut préciser :

La phase : c'est une grandeur, une taille, une quantité relative à une partie des données qui représente un sous-problème. Les résultats d'une phase doivent être déductibles *uniquement* à partir des résultats de la phase précédente.

L'étape : elle correspond à une instance donnée du sous-problème considéré au cours d'une phase.

La décision : c'est une question à laquelle il faut répondre pour l'étape considérée. La réponse à cette question constitue le résultat à obtenir au cours de cette étape.

La formule de récurrence : elle définit le moyen de répondre à la question décisionnelle ; autrement dit, c'est une méthode de déduction du résultat à partir des résultats obtenus lors de la phase précédente pour des sous-problèmes de l'étape considérée.

À partir de cela, il est possible de formuler l'algorithme sous la forme d'une fonction récursive, en précisant quels sont les cas de base.

Il est possible de distinguer deux types d'algorithmes de programmation dynamique :

Forward Les solutions des cas de base sont d'abord calculées ; on en déduit les solutions des problèmes de taille intermédiaire, jusqu'à revenir au problème initial.

Backward Tous les sous-problèmes du problème initial sont énumérés ; on fait la même chose avec tous les sous-problèmes, jusqu'à aboutir au cas de base.

Complexité

La complexité au pire cas d'un algorithme de programmation dynamique dépend de la fonction récursive qui définit le problème. Dans le cas général, on peut seulement dire qu'elle est en $\mathcal{O}^*(\alpha^n)$, avec α une constante positive, le plus souvent entière.

3.2.3 *Sort & Search*

Principe

Les algorithmes de type *Sort & Search*, aussi appelé *Tri & Recherche* en français, suivent tous cette trame générale :

1. Séparation des données en entrée de l'algorithme en deux parties P_1 et P_2
2. Pré-traitement de P_1 :
   (a) Énumération des sous-parties $p_1(j)$
   (b) TRI des sous-parties selon un ou plusieurs critères $C_1^{(1)}$, $C_1^{(2)}$, ...
3. Pré-traitement de P_2 :
   (a) Énumération des sous-parties $p_2(k)$
   (b) TRI des sous-parties selon un ou plusieurs critères $C_2^{(1)}$, $C_2^{(2)}$, ...
4. Pour chaque solution partielle $p_1(j) \in P_1$, RECHERCHE de la meilleure (au sens de la fonction objectif associée au problème) solution partielle $p_2(k_j) \in P_2$ pouvant lui être associée (selon les contraintes du problème) pour former une solution complète
5. Sélection de la meilleure (au sens de la fonction objectif associée au problème) solution $(p_1(j), p_2(k_j))$ trouvée

Cela implique notamment que les solutions du problème étudié doivent être *décomposables* : il faut pouvoir les séparer, et être en mesure de reconstituer une solution complète, à partir des solutions partielles qui la composent, en temps polynomial.

Les critères de tri, ainsi que les méthodes d'association de solutions partielles utilisées pendant la phase de recherche, découlent de propriétés spécifiques liées au problème étudié : les données sont triées au préalable pour accélérer autant que possible la phase de recherche.

Complexité

La complexité au pire cas d'un algorithme de type *Sort & Search* dépend à la fois des tris et des recherches à effectuer. Dans le cas général, on peut seulement dire qu'elle est en $\mathcal{O}^*(\alpha^n)$, avec α une constante positive réelle.

3.2.4 Décomposition

Principe

Ce type d'algorithmes est assez proche des algorithmes de programmation dynamique, en ce sens que, pour trouver une politique optimale, on étudie des sous-politiques optimales.

La différence se situe à ce niveau : dans un algorithme de programmation dynamique, la même fonction récursive est utilisée pour trouver la meilleure solution pour le problème et ses sous-problèmes, alors que dans un algorithme de décomposition, un algorithme exponentiel (de n'importe quel type) est utilisé pour trouver les sous-politiques optimales.

Un algorithme de décomposition est donc défini par les éléments suivants :

1. une manière d'énumérer des solutions partielles ;
2. un algorithme pour trouver les meilleures solutions partielles ;
3. une méthode de reconstitution d'une solution complète à partir de solutions partielles (qui fonctionne en temps polynomial).

Cela implique donc que les solutions soient *décomposables*.

Complexité

La complexité au pire cas d'un algorithme de décomposition dépend de la fonction récursive qui définit le problème. Dans le cas général, on peut seulement dire qu'elle est en $\mathcal{O}^*(\alpha^n)$, avec α une constante positive réelle.

3.2.5 *Branch & Reduce*

Principe

Dans un algorithme de type *Branch & Reduce*, l'espace des solutions est scindé en plusieurs partitions, selon une méthode récursive à définir en fonction du problème. Comme pour le *Branch & Bound*, cela permet de visualiser la progression de l'algorithme en utilisant un modèle de graphe dont les nœuds sont les partitions (*Branch*).

De même, l'objectif est de ne pas avoir à parcourir l'ensemble de l'arbre. Pour cela, on utilise des règles de réduction (*Reduce*) : il s'agit de propriétés portant sur les caractéristiques des partitions, qui indiquent s'il est possible ou non de trouver une solution optimale dans un sous-arbre dont la racine est connue. Contrairement aux bornes calculées dans un algorithme *Branch & Bound*, les règles de réduction d'un algorithme *Branch & Reduce* peuvent être utilisées avant même de commencer à parcourir l'ensemble des feuilles, ce qui permet de réduire la taille de cet ensemble dès le départ.

Complexité

La complexité au pire cas d'un algorithme de type *Branch & Reduce* dépend du partitionnement récursif et des règles de réduction. Dans le cas général, on peut seulement dire qu'elle est en $\mathcal{O}^*(\alpha^n)$, avec α une constante positive, le plus souvent entière.

Branch & Reduce vs Problèmes de permutation

Dans ce PFE, on s'intéresse à des problèmes d'ordonnancement, c'est-à-dire des problèmes de permutation : il est important de signaler qu'il est en général difficile de trouver des règles de réduction pour ce type de problème.

3.2.6 Lien avec des problèmes de graphe

Principe

Plutôt que de chercher à concevoir des méthodes de recherches récursives, ou de s'efforcer de trouver des propriétés spécifiques à un problème, il peut être judicieux d'essayer de convertir le problème pour le transformer en un graphe, sur lequel il est possible d'utiliser des algorithmes déjà connus.

On peut penser notamment aux algorithmes suivants :

- Recherche de plus court chemin
- Coloration *(List Coloring Problem)*
- Recherche d'un ensemble de sommets non reliés de taille maximale *(Maximum Independent Set Problem)*
- Problème du voyageur de commerce

Complexité

La complexité dépend alors du type de problème de graphe auquel le problème initial a été ramené. Il faut faire attention au nombre de sommets N et d'arcs/arêtes M générés. Un algorithme de graphe peut avoir une complexité au pire cas polynomiale en fonction de N et M, mais la complexité associée à l'algorithme résolvant le problème de départ deviendra exponentielle si on génère un nombre de nœuds exponentiel.

3.3 Illustration sur des problèmes d'ordonnancement

3.3.1 Résolution du problème $1|dec|\sum f_i$ par la programmation dynamique

Présentation du problème $1|dec|\sum f_i$

- Données
  - 1 machine, n jobs
  - Pour chaque job $i = 1..n$
    - p_i = durée d'exécution du job i
    - f_i = fonction coût associée au job i
- Variables
  - Pour chaque job $i = 1..n$
    - C_i = date réelle de fin d'exécution du job i
- Hypothèse
  - Le problème est **décomposable**. *(pas de temps mort sur la machine)*
- Objectif
  - Trouver un ordonnancement σ qui minimise la somme des coûts :

$$MIN\left(\sum_{i=1}^{n} f_i(C_i(\sigma))\right)$$

Résolution par la programmation dynamique

- Cet algorithme travaille sur des sous-ensembles de jobs S.
- Récurrence : fonction Opt
  - $Opt[\emptyset] = 0$
  - $Opt[S] = \min_{t \in S}\left\{Opt[S - \{t\}] + f_t\left(\sum_{i \in S} p_i\right)\right\}$
- Initialisation de l'algorithme avec l'ensemble contenant tous les jobs : calcul de $Opt[\{1, \cdots, n\}]$

Complexité de l'algorithme

Hypothèse Le temps de calcul de la fonction f_t pour tout job t est polynomial : $\mathcal{O}^*(1)$.

Pour chaque sous-ensemble de jobs S, $Opt[S]$ est calculé en $\mathcal{O}^*(1)$. Puisqu'il existe 2^n sous-ensembles de jobs, la complexité de l'algorithme est donc en $\mathcal{O}^*(2^n)$

3.3.2 Résolution du problème $P_2||C_{max}$ par un algorithme *Sort & Search*

Présentation du problème $P_2||C_{max}$

- Données
  - 2 machines identiques en parallèle, n jobs
  - Pour chaque job $i = 1..n$
    - ▷ p_i = durée d'exécution du job i sur une machine
- Variables
  - Pour chaque job $i = 1..n$
    - ▷ C_i = date réelle de fin d'exécution du job i
  - $C_{max} = \max_{1 \leqslant i \leqslant n} C_i$
- Objectif
  - Traiter l'ensemble des jobs le plus rapidement possible :

$$MIN\,(C_{max})$$

Résolution par un algorithme de type *Sort & Search*

- Partitionnement de l'ensemble des jobs
  - I_1 contient les jobs 1 à $\lfloor \frac{n}{2} \rfloor$. → Construction d'une table T_1 :
    - ▷ s_1^j = ensemble des jobs de I_1 traités par la machine 1
    - ▷ $\bar{s}_1^j = I_1 \backslash s_1^j$ = ensemble des jobs de I_1 traités par la machine 2
  - I_2 contient les jobs $\lfloor \frac{n}{2} \rfloor + 1$ à n. → Construction d'une table T_2 :
    - ▷ s_2^k = ensemble des jobs de I_2 traités par la machine 1
    - ▷ $\bar{s}_2^k = I_2 \backslash s_2^k$ = ensemble des jobs de I_2 traités par la machine 2

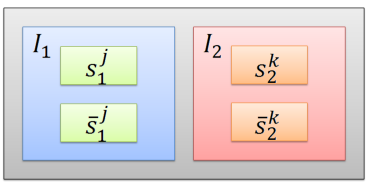


FIGURE 3.1 – $P_2||C_{max}$, *Sort & Search* : partitionnement de l'ensemble des jobs

- Constitution d'une séquence
  - Jobs traités par la machine 1
    - ▷ Tous les jobs dans s_1^j et s_2^k
  - Jobs traités par la machine 2

▷ Tous les jobs dans $\bar{s}_1^j$ et $\bar{s}_2^k$

- Par conséquent : $C_{max} = \max\left(\sum_{i\in s_1^j} p_i + \sum_{i\in s_2^k} p_i \ , \ \sum_{i\in \bar{s}_1^j} p_i + \sum_{i\in \bar{s}_2^k} p_i\right)$

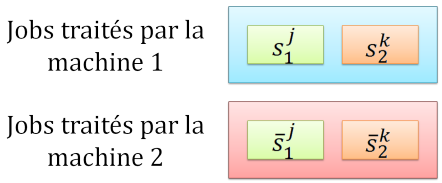


FIGURE 3.2 – $P_2||C_{max}$, *Sort & Search* : reconstitution d'une solution

→ Étant donné un sous-ensemble $s_1^j \in T_1$, comment choisir le sous-ensemble $s_2^k \in T_2$ à lui associer ?

- Préparation des tables
  - Pour chaque $s_1^j \in T_1$
    - ▷ Calcul de $P(s_1^j) = \sum_{i\in s_1^j} p_i$
    - ▷ Calcul de $P(\bar{s}_1^j) = \sum_{i\in \bar{s}_1^j} p_i$
  - Pour chaque $s_2^k \in T_2$
    - ▷ Calcul de $P(s_2^k) = \sum_{i\in s_2^k} p_i$
    - ▷ Calcul de $P(\bar{s}_2^k) = \sum_{i\in \bar{s}_2^k} p_i$
    - → **Tri** par $\left(P(\bar{s}_2^k) - P(s_2^k)\right)$ décroissant (définit la position k)
    - ▷ Calcul de $P_{min}^d(s_2^k) = \min_{\ell\geqslant k}(P(s_2^\ell))$ (d pour « droite »)
    - ▷ Calcul de $P_{min}^g(s_2^k) = \min_{\ell\leqslant k}(P(\bar{s}_2^\ell))$ (g pour « gauche »)
- Mise en correspondance des tables T_1 et T_2
  - Pour chaque $s_1^j \in T_1$
    - ▷ **Recherche** de $s_2^k \in T_2$, tel que l'indice k vérifie :
      $k = \arg\min(u \in T_2 \mid P(s_1^j) - P(\bar{s}_1^j) \geqslant P(\bar{s}_2^u) - P(s_2^u))$
      → $C_{max} = C_{max}^d = P(s_1^j) + P_{min}^d(s_2^k)$
    - ▷ **Recherche** de $s_2^\ell \in T_2$, tel que l'indice ℓ vérifie :
      $\ell = \arg\max(u \in T_2 \mid P(s_1^j) - P(\bar{s}_1^j) \leqslant P(\bar{s}_2^u) - P(s_2^u))$
      → $C_{max} = C_{max}^g = P(\bar{s}_1^j) + P_{min}^g(s_2^\ell)$
    - ▷ s_1^j est associé à $s_2^{(min)} \in T_2$:
      Soit $s_2^d \in T_2$ une séquence pour laquelle $P_{min}^d(s_2^k)$ est atteint.
      Soit $s_2^g \in T_2$ une séquence pour laquelle $P_{min}^g(s_2^\ell)$ est atteint.
      $$s_2^{(min)} = \begin{cases} s_2^d & \text{si } C_{max}^d \leqslant C_{max}^g \\ s_2^g & \text{sinon} \end{cases}$$
- Extraction d'une solution optimale
  - Association de $s_1^j \in T_1$ et $s_2^{(min)} \in T_2$ qui minimise le C_{max}.

Complexité de l'algorithme

- La table T_1 contient $2^{\lfloor n/2 \rfloor}$ éléments : la complexité associée à sa construction est donc en $\mathcal{O}^*(\sqrt{2}^n)$ (temps et espace).
- La table T_2 contient $2^{\lceil n/2 \rceil}$ éléments triés : la complexité associée à sa construction est donc en $\mathcal{O}^*(2^{n/2} + 2^{n/2}\log(2^{n/2})) = \mathcal{O}^*(\sqrt{2}^n)$ (temps et espace).
- Étant donné $s_1^j \in T_1$, la recherche de $s_2^d \in T_2$ et $s_2^g \in T_2$, sachant que T_2 est triée, revient à faire une dichotomie, en $\mathcal{O}^*(\log(2^{n/2})) = \mathcal{O}(n) = \mathcal{O}^*(1)$ (temps).
- La recherche est effectuée pour tous les éléments de T_1, donc en $\mathcal{O}^*(\sqrt{2}^n)$ (temps).

Par conséquent, la complexité de cet algorithme est en $\mathcal{O}^*(\sqrt{2}^n)$.

3.3.3 Résolution du problème $P_4||C_{max}$ par un algorithme de décomposition

Présentation du problème $P_4||C_{max}$

- Données
  - 4 machines identiques en parallèle, n jobs
  - Pour chaque job $i = 1..n$
    - ▷ p_i = durée d'exécution du job i sur une machine
- Variables
  - Pour chaque job $i = 1..n$
    - ▷ C_i = date réelle de fin d'exécution du job i
  - $C_{max} = \max\limits_{1 \leqslant i \leqslant n} C_i$
- Objectif
  - Traiter l'ensemble des jobs le plus rapidement possible :

$$MIN\,(C_{max})$$

Résolution par un algorithme de décomposition

- Définition de 2 ensembles de machines
  - $\mathcal{M}_1$ contient les machines 1 et 2.
  - $\mathcal{M}_2$ contient les machines 3 et 4.
- Énumération de toutes les affectations possibles des jobs : soit sur $\mathcal{M}_1$, soit sur $\mathcal{M}_2$
- Pour chacune des affectations :
  - Utilisation de l'algorithme *Sort & Search* vu pour le problème $P2||C_{max}$ pour résoudre le sous-problème sur $\mathcal{M}_1$
  - Utilisation de l'algorithme *Sort & Search* vu pour le problème $P2||C_{max}$ pour résoudre le sous-problème sur $\mathcal{M}_2$
- Une solution optimale minimise $C_{max} = \max\left(C_{max}^{(\mathcal{M}_1)}\ ,\ C_{max}^{(\mathcal{M}_2)}\right)$.

Complexité de l'algorithme

Rappel La complexité de l'algorithme résolvant le $P_2||C_{max}$ est $\mathcal{O}^*(\sqrt{2}^n)$ pour une instance contenant n jobs.

Dans le cas où k jobs sont placés sur l'ensemble de machines $\mathcal{M}_1$, les $n-k$ jobs restants sont placés sur l'ensemble de machines $\mathcal{M}_2$. La résolution du premier sous-problème se fait en $\mathcal{O}^*(\sqrt{2}^k)$; la résolution du second sous-problème se fait en $\mathcal{O}^*(\sqrt{2}^{(n-k)})$.

La complexité globale du $P_2||C_{max}$ est donc :

$$\mathcal{O}^*\left(\sum_{k=0}^{n}\binom{n}{k}(\sqrt{2}^k+\sqrt{2}^{n-k})\right)=\mathcal{O}^*\left(2\times\sum_{k=0}^{n}\binom{n}{k}\sqrt{2}^k 1^{n-k}\right)=\mathcal{O}^*\left((\sqrt{2}+1)^n\right)$$

3.3.4 Résolution du problème $F_2||C^k_{max}$ par un algorithme *Branch & Reduce*

Présentation du problème $F_2||C^k_{max}$

- Données
  - 2 machines (atelier organisé en *flowshop*), n jobs
  - Pour chaque job $i = 1..n$
    - ▷ $p_{i,j}$ = durée d'exécution du job i sur la machine j
  - k = nombre de jobs à traiter le plus rapidement possible $(1 \leqslant k \leqslant n)$
- Variables
  - Pour chaque job $i = 1..n$
    - ▷ C_i = date réelle de fin d'exécution du job i
  - $C^k_{max} = \max_{i \in E_k} C_i$ où E_k est l'ensemble des k premiers jobs traités
- Objectif
  - Traiter les k premiers jobs le plus rapidement possible :

$$MIN\left(C^k_{max}\right)$$

Autre formulation du problème : $F_2|d_i = d, d \text{ unknown}, \sum U_i = \epsilon|d$ (avec $\epsilon = n-k$)

- Données
  - 2 machines (atelier organisé en flowshop), n jobs
  - Pour chaque job $i = 1..n$, pour chaque machine $j = 1..m$
    - ▷ $p_{i,j}$ = durée d'exécution du job i sur la machine j
  - ϵ = nombre de jobs en retard $(1 \leqslant \epsilon \leqslant n)$ $\rightarrow \epsilon = n-k$
- Variables
  - d = date de fin d'exécution « souhaitée » (la même pour tous les jobs)
  - Pour chaque job $i = 1..n$
    - ▷ C_i = date réelle de fin d'exécution du job i
    - ▷ $U_i = \begin{cases} 1 & \text{si le job } i \text{ est en retard} \quad (C_i > d) \\ 0 & \text{sinon} \end{cases}$
- Contrainte
  - Le nombre de jobs en retard est imposé : $\sum_{i=1}^{n} U_i = \epsilon$
- Objectif
  - Minimiser la date de fin d'exécution « souhaitée » :

$$MIN\,(d)$$

Résolution par un algorithme de type *Branch & Reduce*

- Construction d'un arbre binaire complet de profondeur n
  - Pour chaque nœud de profondeur $i-1$ $(1 \leqslant i \leqslant n)$
    - ▷ Dans le fils gauche, le job i est en avance.
    - ▷ Dans le fils droit, le job i est en retard.

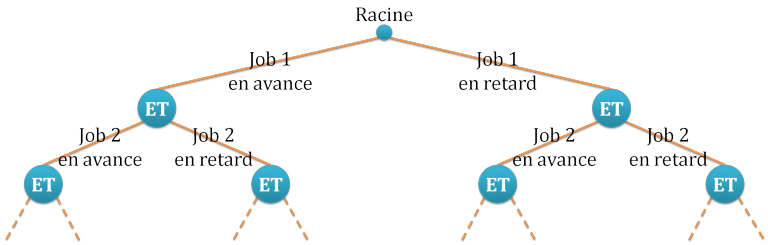


FIGURE 3.3 – $F_2||C^k_{max}$, *Branch & Reduce* : représentation des premiers niveaux de l'arbre

- Élimination des nœuds pour lesquels le nombre de jobs en retard est supérieur à $\epsilon = n - k$
- Pour chaque feuille restante :
  - Calcul de $C^k_{max} = d$ (règle de Johnson appliquée sur l'ensemble des k jobs en avance)
- Une solution optimale correspond à une feuille pour laquelle la valeur de C^k_{max} est minimale.

Complexité de l'algorithme

Notons $\mathcal{C}(n, \epsilon)$ la complexité associée à la résolution d'une instance à n jobs dont ϵ en retard. Le schéma de branchement est tel que :

- $\mathcal{C}(0, 0) = \mathcal{O}(1)$ (complexité constante au niveau des feuilles de l'arbre)
- $\mathcal{C}(n, \epsilon) = \mathcal{C}(n-1, \epsilon) + \mathcal{C}(n-1, \epsilon-1)$ (exploration des deux fils pour chaque nœud)

Par conséquent : $\mathcal{C}(n, \epsilon) = \binom{n}{\epsilon}$

Soit $\lambda = \dfrac{\epsilon}{n} (\lambda \in [0; 1])$. En utilisant la formule de Stirling, on obtient le résultat suivant :

$$\mathcal{C}(n, \epsilon) = f(\lambda)^n \qquad \text{avec} \qquad f(\lambda) = \left(\frac{1}{\lambda}\right)^{\lambda} \left(\frac{1}{1-\lambda}\right)^{1-\lambda}$$

Remarque Une symétrie apparaît dans cette expression : $\mathcal{C}(n, \epsilon) = \mathcal{C}(n, n - \epsilon)$. Cela corrobore le fait que fixer le nombre de jobs en retard équivaut à fixer le nombre de jobs en avance (même complexité).

3.3.5 Résolution du problème *Interval Scheduling* utilisant un graphe

Présentation du problème *Interval Scheduling*

- Données
  - m machines identiques en parallèle, n jobs
  - Pour chaque job $i = 1..n$
    - ▷ $I_i = [r_i, \tilde{d}_i]$ = intervalle de temps sur le lequel le job i doit être exécuté

- ▷ M_i = ensemble des machines sur lesquelles le job i peut être exécuté

- Objectif
  - Trouver une répartition des n jobs sur les m machines qui respecte les contraintes imposées par :
    - ▷ les intervalles de temps $\{I_i\}_{1 \leqslant i \leqslant n}$
    - ▷ les ensembles de machines $\{M_i\}_{1 \leqslant i \leqslant n}$
- Remarque
  - Il s'agit non pas d'un problème d'optimisation, mais d'un problème de décision : existe-t-il une solution réalisable ?

Résolution basée sur la conversion du problème en un graphe non-orienté coloré

- Conversion des données en graphe $G = (V, E)$
  - Sommets (= jobs)
    - ▷ $\forall i = 1..n \qquad i \in V$
  - Couleurs (= machines)
    - ▷ $\forall i = 1..n \qquad C_i = M_i$: ensemble des couleurs utilisables pour le sommet i
  - Arcs non-orientés
    - ▷ Deux sommets relatifs à des jobs différents sont reliés si leurs intervalles de temps d'exécution se chevauchent :

$$\begin{matrix} \forall i_1 \in V \\ \forall i_2 \in V \end{matrix} \qquad i_1 \neq i_2 \text{ et } I_{i_1} \cap I_{i_2} \neq \emptyset \quad \Rightarrow \quad (i_1, i_2) \in E$$

- Recherche d'une coloration du graphe telle que chaque sommet est colorié d'une couleur différente de celle des sommets adjacents.
- Il existe une solution réalisable *si et seulement si* une telle coloration existe.

Complexité de l'algorithme

Soit un graphe comportant N sommets : la recherche d'une coloration du graphe telle que chaque sommet est colorié d'une couleur différente de celle des sommets adjacents est un problème pouvant être résolu $\mathcal{O}^*(2^N)$.

Ici, le nombre de sommets N générés lors de la conversion du problème *Interval Scheduling* en graphe est égal au nombre de jobs n. Par conséquent, la complexité de l'algorithme est en $\mathcal{O}^*(2^n)$.

Étude approfondie de divers algorithmes exponentiels

4.1 Présentation du problème étudié : le $F_3||C_{max}$

4.1.1 Définition

Le $F_3||C_{max}$ est un problème d'ordonnancement de type *flowshop* :

- n tâches (aussi appelées jobs) sont à réaliser.
- Chacune de ces tâches requiert le passage par $m = 3$ machines successives.
- La *gamme* de fabrication est la même pour toutes les tâches : il faut d'abord passer sur la machine 1, puis sur la machine 2, et enfin sur la machine 3.

Chaque tâche est caractérisée par son temps de passage sur chacune des machines : on notera $p_{i\,j}$ la durée de traitement du job j sur la machine i.

Les contraintes suivantes s'appliquent :

- Chaque job ne peut être traité que par une seule machine à la fois.
- Chaque machine ne peut traiter qu'un seul job à la fois.
- Le traitement d'un job sur une machine ne peut pas être interrompu une fois qu'il est commencé (non préemption).

L'objectif est de minimiser le temps total nécessaire à la réalisation de l'ensemble des tâches, noté C_{max}.

4.1.2 Propriétés

Le problème $F_3||C_{max}$ est un cas particulier du problème $F||C_{max}$, pour lequel le nombre de machines m est quelconque.

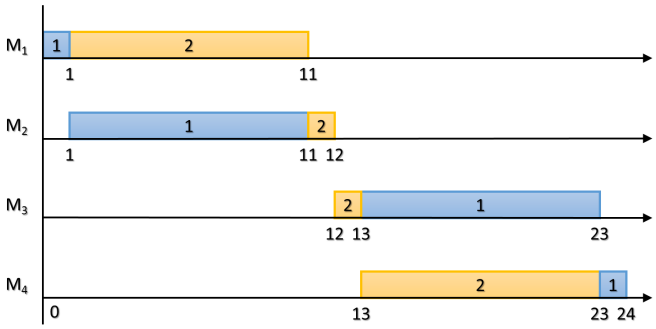
- Lorsque $m = 1$, le problème devient trivial : toute permutation des jobs est optimale (la durée minimale nécessaire à la réalisation de l'ensemble des tâches est égale à la somme des durées de traitement sur l'unique machine).
- Lorsque $m = 2$, la règle de Johnson permet de trouver des solutions optimales en temps polynomial.
  1. Partitionnement de l'ensemble des jobs : $S_1 = \{j \,/\, p_{1\,j} \leqslant p_{2\,j}\}$, $S_2 = \{j \,/\, p_{1\,j} > p_{2\,j}\}$
  2. Tri des jobs de S_1 par valeurs de $p_{1\,j}$ croissantes (règle $SPT(1)$: « *Shortest Processing Time* »)
  3. Tri des jobs de S_2 par valeurs de $p_{2\,j}$ décroissantes (règle $LPT(2)$: « *Largest Processing Time* »)
  4. Concaténation des jobs de S_1 selon l'ordre $SPT(1)$ suivis des jobs de S_2 selon l'ordre $LPT(2)$

  La complexité de cet algorithme est donc en $\mathcal{O}(n \ln(n))$.

- Lorsque $m \geqslant 3$, le problème devient NP-difficile au sens fort.
- Lorsque $m \leqslant 3$, il existe toujours des solutions optimales sous la forme de permutations : dans ce type de solution, tous les jobs sont traités dans le même ordre sur toutes les machines. Ce n'est plus le cas à partir de $m = 4$ machines.

  Exemple Pour l'instance suivante du $F_4||C_{max}$, la solution optimale est obtenue en traitant le job 1 puis le job 2 sur $\mathcal{M}_1$ et $\mathcal{M}_2$, et en traitant le job 2 puis le job 1 sur $\mathcal{M}_3$ et $\mathcal{M}_4$.

Job ID	1	2
$\mathcal{M}_1$	1	10
$\mathcal{M}_2$	10	1
$\mathcal{M}_3$	10	1
$\mathcal{M}_4$	1	10



Remarque Les algorithmes présentés ensuite sont restreints à la recherche de permutations optimales.

4.1.3 Modélisation linéaire

Données

- n jobs, $m = 3$ machines
- p_{ij} = durée de traitement du job j $(1 \leqslant j \leqslant n)$ sur la machine i $(1 \leqslant i \leqslant m)$

Variables

- $x_{kj} = 1$ si le job j est traité en position k $(1 \leqslant k \leqslant n)$, 0 sinon
- C_{ik} = date de fin au plus tôt du job traité en position k sur la machine i

Contraintes

Les $\{x_{kj}\}_{k,j}$ représentent une permutation.

$$\forall j \qquad \sum_k x_{kj} = 1$$

$$\forall k \qquad \sum_j x_{kj} = 1$$

Contraintes relatives au *flowshop*

$$C_{11} = \sum_j x_{1j} p_{1j}$$

$$\begin{cases} \forall i \geqslant 2 \\ \forall k \end{cases} \qquad C_{ik} \geqslant C_{(i-1)k} + \sum_j x_{kj} p_{ij}$$

$$\begin{cases} \forall i \\ \forall k \geqslant 2 \end{cases} \qquad C_{ik} \geqslant C_{i(k-1)} + \sum_j x_{kj} p_{ij}$$

Objectif Minimiser $C_{mn} = C_{max}$

4.2 Algorithme de type programmation dynamique, avec élimination de solutions dominées

4.2.1 Présentation

Le premier algorithme implémenté est un algorithme de type « programmation dynamique fg, dans lequel les séquences ordonnant les jobs sont construites du début vers la fin ; tout au long de l'algorithme, certaines séquences sont éliminées grâce à une condition de dominance.

Séquences de jobs partielles

Soit une instance contenant n jobs. On appelle séquence partielle, toute permutation σ d'un sous-ensemble S de jobs (représentés par leur ID) de l'instance. On note $C_i(\sigma)$ la date à laquelle tous ces travaux peuvent être terminés au plus tôt sur la machine i.

La séquence partielle vide est notée ε : elle n'ordonne aucun job ($S = \emptyset$). Comme les machines ne sont pas utilisées, $C_i(\varepsilon) = 0$ pour toute machine i.

Il est possible d'ajouter un job j à la fin d'une séquence σ ($j \notin S$) pour former une nouvelle séquence $\mu = \sigma|j$:

$$\begin{cases} & C_1(\sigma|j) = C_1(\sigma) + p_{1\,j} \\ \forall i \geqslant 2 & C_i(\sigma|j) = \max(C_{i-1}(\sigma|j), C_i(\sigma)) + p_{i\,j} \end{cases}$$

Relation de dominance entre séquences de jobs partielles

Soit S un sous-ensemble de jobs. On souhaite mettre les jobs de S au début de l'ordonnancement. Soit σ_1 et σ_2 deux permutations des jobs qui appartiennent à S. Soit τ une permutation des jobs restants (qui n'appartiennent pas à S).

On dit que σ_1 **domine** σ_2 lorsque $C_i(\sigma_1) \leqslant C_i(\sigma_2)$ pour toute machine i. Dans le cas particulier du $F_3||C_{max}$, le nombre de machines est égal à 3. La **condition de dominance** entre séquences partielles s'écrit alors :

$$\sigma_1 \preccurlyeq \sigma_2 \qquad \Leftrightarrow \qquad \text{et} \begin{cases} C_2(\sigma_1) \leqslant C_2(\sigma_2) \\ C_3(\sigma_1) \leqslant C_3(\sigma_2) \end{cases}$$

Remarque Puisqu'il n'y a jamais de temps mort sur la première machine, comme σ_1 et σ_2 ordonnent les mêmes jobs : $C_1(\sigma_1) = C_1(\sigma_2)$

Cette condition de dominance implique la propriété suivante : $C_3(\sigma_1|\tau) \leqslant C_3(\sigma_2|\tau)$. Cela signifie que les solutions commençant par σ_1 se finissent forcément plus tôt (ou à la même date) que les solutions commençant par σ_2. Par conséquent, la séquence partielle σ_2 peut être ignorée lors de la recherche d'une solution optimale.

Schéma de programmation dynamique

L'algorithme de programmation dynamique est défini ainsi :

Phase La taille t des sous-ensembles de jobs à considérer

Étape L'un des sous-ensembles de jobs S de taille t

Décision Parmi l'ensemble E des séquences ordonnant les jobs de S, quelles sont celles qui ne sont dominées par aucune autre séquence de E ?

Récurrence Il s'agit de trouver ces séquences, *uniquement* à partir de celles trouvées pour les ensembles de jobs de taille $t-1$.

Soit Opt la fonction qui, à un sous-ensemble de jobs S, associe l'ensemble des séquences non-dominées ordonnant les jobs dans S.

- La phase 0 (initiale) consiste à associer la permutation vide ε à l'ensemble de jobs vide.

$$Opt(\emptyset) = \{\varepsilon\}$$

- Pour la phase t, on énumère les sous-ensembles de t jobs parmi n. Pour chacun de ces sous-ensembles, on génère les séquences correspondantes :
  1. On enlève un job de l'ensemble. (t possibilités)
  2. On récupère les séquences non-dominées (générées lors de la phase $t-1$) qui ordonnent les jobs restants.
  3. On ajoute le job précédemment retiré à la fin de toutes ces séquences.

  Une fois que toutes les séquences associées au sous-ensemble de jobs ont été générées, on élimine les séquences dominées, pour conserver seulement le front de Pareto.

$$Opt(S) = ParetoFront\left(\bigcup_{j\in S}\left(\bigcup_{\sigma\in Opt(S\setminus\{j\})}\{\sigma|j\}\right)\right)$$

Une solution optimale est une séquence $\sigma^* \in Opt(\{1,\cdots,n\})$ pour laquelle $C_3(\sigma^*)$ est minimal.

4.2.2 Complexité

Pour chaque sous-ensemble de t jobs, le front de Pareto associé contient au plus 2^t points.

Le front de Pareto pour un ensemble de taille t est obtenu par fusion de t fronts de Pareto contenant au plus 2^{t-1} points : en effet, chacun de ces fronts est associé à un ensemble de taille $t-1$ (obtenu en enlevant l'un des t jobs).

La complexité au pire cas de l'extraction d'un front de Pareto sur un ensemble de taille K est $\mathcal{O}(K\ln(K))$.

Par conséquent, la complexité au pire cas associée à l'obtention d'un front de Pareto pour un ensemble de jobs de taille t est en $\mathcal{O}^*(2^t)$:

$$\mathcal{O}\left(t\,2^{t-1}\,\ln(t\,2^{t-1})\right) = \mathcal{O}\left(t\,2^{t-1}\,[\ln(t)+(t-1)\,\ln(2)]\right) = \mathcal{O}^*(2^t)$$

On en déduit (par la formule du binôme de Newton) que la complexité au pire cas de cet algorithme de programmation dynamique est en $\mathcal{O}^*(3^n)$:

$$\mathcal{C}(n) = \mathcal{O}^*\left(\sum_{t=1}^{n}\binom{n}{t}2^t\right) = \mathcal{O}^*(3^n)$$

4.2.3 Implémentation

Dans cet algorithme, à tout sous-ensemble de jobs S est associé le front de Pareto $Opt(S)$. C'est pourquoi un conteneur associatif, une `std::map`, a été utilisé :

- Les *clés* sont les sous-ensembles de jobs, représentés par leurs signatures.
- Les *valeurs* sont les fronts de Pareto.

Définition de la signature d'un sous-ensemble de jobs S

Il s'agit d'une suite de 0 et de 1, codée sous la forme d'un `std::vector<bool>`, où chaque symbole correspond à l'un des jobs de l'instance :

- Si le j^e symbole vaut 1, alors le job j appartient à S.
- Si le j^e symbole vaut 0, alors le job j n'appartient pas à S.

Codage des valeurs de la `std::map`

Deux implémentations distinctes ont été testées :

1. La première consiste à stocker des séquences partielles dans un `std::vector<Sequence>`, puis à éliminer les séquences dominées grâce à un foncteur `Pareto::Extractor` (voir schéma page 26).
2. La seconde consiste à utiliser un conteneur « intelligent », `Pareto::Set` (voir schéma page 26). À chaque tentative d'insertion d'une nouvelle séquence σ, le conteneur procède comme suit :
   - Recherche d'une séquence appartenant au `Pareto::Set` qui domine σ
   - Si aucune séquence n'est trouvée : ajout de σ dans le `Pareto::Set`, et suppression des séquences appartenant au `Pareto::Set` dominées par σ (éventuellement aucune)

Gestion des phases de l'algorithme de programmation dynamique

L'algorithme a été implémenté en *forward* : la phase 0 est réalisée en premier, puis la phase 1, jusqu'à la phase n. Pour une phase donnée, une `std::map` est utilisée. Comme les résultats de la phase t sont obtenus *uniquement* à partir des résultats de la phase $t-1$, le programme ne conserve en mémoire que deux `std::map` (une pour la phase t, une autre pour la phase $t-1$).

4.3 Proposition d'une variante pour l'algorithme de programmation dynamique

Cette partie du rapport décrit un algorithme proposé par l'étudiant, qui est directement inspiré de l'algorithme précédent. Cet algorithme n'a finalement pas été retenu pour l'implémentation.

4.3.1 Rappels sur la construction de solution dans l'algorithme précédent

Dans l'algorithme de programmation dynamique présenté auparavant, l'idée directrice est la suivante : construire des séquences de job du début vers la fin, en ajoutant les jobs un par un, tout en éliminant au fur et à mesure des séquences partielles dominées. Chaque séquence non-dominée correspond à un préfixe utilisable pour une solution (séquence ordonnançant tous les jobs).

Ici, l'idée est de considérer des séquences dont la taille va *doubler* d'une itération à la suivante : plutôt que d'ajouter les jobs les uns après les autres, les séquences sont concaténées directement. Pour chaque séquence, il faudra déterminer si elle peut être utilisée comme *préfixe* et/ou *suffixe* d'une solution optimale (extension de la condition de dominance).

4.3.2 Description de la variante

Remarque sur la concaténation de séquences

Pour le problème $F||C_{max}$, le calcul des dates de début et de fin sur toutes les machines pour une nouvelle séquence $\sigma|\tau$, obtenue par concaténation de deux séquences partielles σ et τ (qui ordonnent des jobs différents), est un traitement qui peut être réalisé au pire cas en temps pseudo-polynomial : $\mathcal{O}(mn)$.

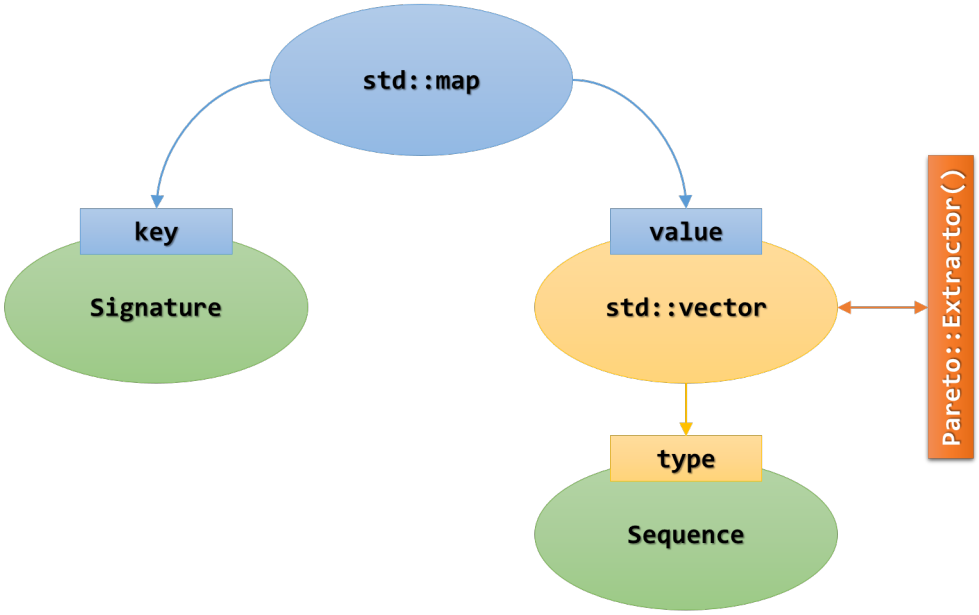


FIGURE 4.1 – Algorithme de programmation dynamique : structure de données (implémentation 1)

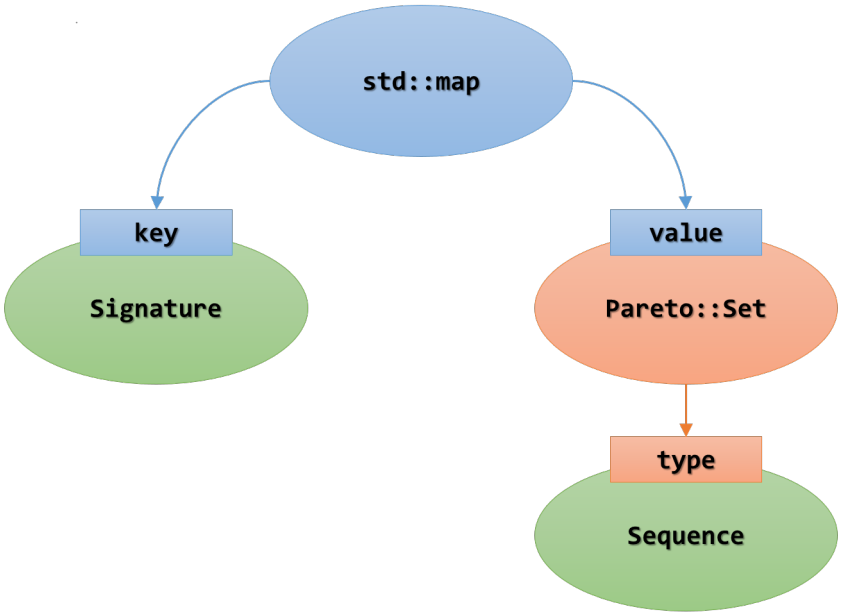


FIGURE 4.2 – Algorithme de programmation dynamique : structure de données (implémentation 2)

Pour le $F_3||C_{max}$, le nombre de machines m est fixée à 3 : la complexité devient linéaire en fonction du nombre de jobs : $\mathcal{O}(n)$.

Une méthode a été proposée par l'étudiant, pour réaliser ce traitement en temps constant ($\mathcal{O}(1)$). Cependant, cette méthode était assez complexe (considération de quatre sous-problèmes simultanément, obtenus en réduisant parfois le nombre de machines considérées, et en réalisant les traitements des jobs sur les machines soit au plus tôt soit au plus tard), et sa validité n'a pas pu être prouvée : c'est pourquoi elle n'a pas été reportée dans ce document.

Quoi qu'il en soit, la complexité au pire cas de ce traitement peut être bornée ainsi : $\mathcal{O}^*(1)$.

Dominance entre séquences : préfixes et suffixes

Soit S un sous-ensemble de jobs. On souhaite mettre les jobs de S soit tout au début (préfixe) soit tout à la fin (suffixe) de l'ordonnancement. Soit σ_1 et σ_2 deux permutations des jobs qui appartiennent à S. Soit τ une permutation des jobs restants (qui n'appartiennent pas à S).

Étant donné une séquence partielle de jobs σ, on adopte les notations suivantes :

- $C_i(\sigma)$ désigne la date de fin de traitement du job en dernière position sur la machine i, en réalisant tous les traitements le plus tôt possible.
- $B_i(\sigma)$ désigne la date de début de traitement du job en première position sur la machine i, en réalisant tous les traitements le plus tard possible, sans pour autant retarder la date de fin au plus tôt sur la dernière machine ($= C_{max}$).

Parmi les séquences partielles à utiliser comme *préfixe* d'une solution, il est possible d'identifier celles qui peuvent être ignorées en comparant les valeurs des $C_i(\sigma)$ (relation $\preccurlyeq_P$). Il s'agit de la même condition que pour l'algorithme précédent.

$$\sigma_1 \preccurlyeq_P \sigma_2 \quad \Leftrightarrow \quad \text{et} \begin{cases} C_2(\sigma_1) \leqslant C_2(\sigma_2) \\ C_3(\sigma_1) \leqslant C_3(\sigma_2) \end{cases}$$

Symétriquement, parmi les séquences partielles à utiliser comme *suffixe* d'une solution, il est possible d'identifier celles qui peuvent être ignorées en comparant les valeurs des $B_i(\sigma)$ (relation $\preccurlyeq_S$).

$$\sigma_1 \preccurlyeq_S \sigma_2 \quad \Leftrightarrow \quad \text{et} \begin{cases} B_2(\sigma_1) \geqslant B_2(\sigma_2) \\ B_1(\sigma_1) \geqslant B_1(\sigma_2) \end{cases}$$

Schéma de programmation dynamique

L'algorithme de programmation dynamique est défini de la même manière, sauf pour la récurrence :

Phase La taille t des sous-ensembles de jobs à considérer ; le numéro de la phase sera donné par $\lfloor \log_2(t) \rfloor$

Étape L'un des sous-ensembles de jobs S de taille t

Décision Parmi l'ensemble E des séquences ordonnant les jobs de S, quelles sont celles qui ne sont dominées par aucune autre séquence de E ?

Récurrence Il s'agit de trouver ces séquences, *uniquement* à partir de celles trouvées pour les ensembles de jobs de taille $\lfloor \frac{t}{2} \rfloor$.

Pour formaliser la récurrence, trois fonctions sont utilisées : Base, Prefix, Suffix. Pour un sous-ensemble de jobs S donné :

1. $\mathrm{Base}(S)$ désigne un ensemble de séquences partielles ordonnant les jobs de S (*a priori*, pas toutes).
2. $\mathrm{Prefix}(S)$ est un sous-ensemble de $\mathrm{Base}(S)$ qui ne contient que des séquences non dominées au sens de la relation $\preccurlyeq_P$.
3. $\mathrm{Suffix}(S)$ est un sous-ensemble de $\mathrm{Base}(S)$ qui ne contient que des séquences non dominées au sens de la relation $\preccurlyeq_S$.

Deux opérateurs permettant d'extraire des fronts de Pareto sont également utilisés :

1. $\mathrm{PrefixFront}$ pour la relation $\preccurlyeq_P$ (préfixes)
2. $\mathrm{SuffixFront}$ pour la relation $\preccurlyeq_S$ (suffixes)

Voici les relations de récurrence entre tous ces éléments :

- Lors de la phase 0 initiale, pour tous les sous-ensembles de jobs de taille $t = 1$, l'unique séquence correspondante est calculée :

$$\forall j \qquad \mathrm{Base}(\{j\}) = \mathrm{Prefix}(\{j\}) = \mathrm{Suffix}(\{j\}) = \{\bar{j}\}$$

- Dans le cas récurrent, pour un sous-ensemble de jobs S de taille t paire :
  1. On isole la moitié des jobs : $T \subset S$, $|T| = \frac{t}{2}$ ($\frac{t}{2}$ possibilités parmi t).
  2. On sélectionne une séquence $\sigma \in \mathrm{Prefix}(T)$ (au plus $2^{\frac{t}{2}}$ possibilités).
  3. On sélectionne une séquence $\tau \in \mathrm{Suffix}(S \backslash T)$ (au plus $2^{\frac{t}{2}}$ possibilités).
  4. On ajoute $\sigma|\tau$ dans $\mathrm{Base(S)}$.

$$\mathrm{Base}(S) = \bigcup_{T \subset S\,,\,|T|=\frac{|S|}{2}} \bigcup_{\sigma \in \mathrm{Prefix}(T)} \bigcup_{\tau \in \mathrm{Suffix}(S \backslash T)} \{\sigma|\tau\}$$

- Dans le cas récurrent, pour un sous-ensemble de jobs S de taille t impaire :
  1. On isole un job j (t possibilités) : $S \backslash \{j\}$ est un sous-ensemble de jobs de taille paire.
  2. On isole la moitié des jobs restants : $T \subset S \backslash \{j\}$, $|T| = \frac{t-1}{2}$ ($\frac{t-1}{2}$ possibilités parmi t).
  3. On sélectionne une séquence $\sigma \in \mathrm{Prefix}(T)$ (au plus $2^{\frac{t-1}{2}}$ possibilités).
  4. On sélectionne une séquence $\tau \in \mathrm{Suffix}(S \backslash (T \cup \{j\}))$ (au plus $2^{\frac{t-1}{2}}$ possibilités).
  5. On ajoute $\sigma|\bar{j}|\tau$ dans $\mathrm{Base(S)}$.

$$\mathrm{Base}(S) = \bigcup_{j \in S} \bigcup_{T \subset S \backslash \{j\}\,,\,|T|=\frac{|S|-1}{2}} \bigcup_{\sigma \in \mathrm{Prefix}(T)} \bigcup_{\tau \in \mathrm{Suffix}(S \backslash (T \cup \{j\}))} \{\sigma|\tau\}$$

- Dans le cas récurrent, une fois que $\mathrm{Base}(S)$ a été calculé, $\mathrm{Prefix}(S)$ (respectivement $\mathrm{Suffix}(S)$) est calculé en utilisant l'opérateur $\mathrm{PrefixFront}$ (respectivement $\mathrm{SuffixFront}$).

$$\mathrm{Prefix}(S) = \mathrm{PrefixFront}(\mathrm{Base}(S))$$

$$\mathrm{Suffix}(S) = \mathrm{SuffixFront}(\mathrm{Base}(S))$$

Une solution optimale est une séquence $\sigma^* \in \mathrm{Base}(\{1, \cdots, n\})$ pour laquelle $C_3(\sigma^*)$ est minimal.

4.3.3 Complexité

Hypothèse Le nombre de jobs n dans l'instance est une puissance de 2.

Notation Dans la formule ci-dessous, l'indice k désigne le numéro de la phase au cours de laquelle des sous-ensembles de jobs de taille $t = 2^k$ sont considérés (c'est-à-dire $k = \log_2(t)$).

$$\mathcal{C}(n) = \mathcal{O}(n) + \sum_{k=1}^{\log_2(n)} \binom{n}{2^k} \left\{ \left[\binom{2^k}{2^{k-1}} 2^{2^{k-1}} 2^{2^{k-1}} \mathcal{O}^*(1) \right] + \mathcal{O}^*(2^{2^k}) \right\}$$

Remarque La recherche d'un nombre réel α tel que $\mathcal{C}(n) = \mathcal{O}^*(\alpha^n)$ n'a pas abouti, du fait de la complexité de cette expression.

4.4 Algorithme à base de « triplets »

4.4.1 Présentation

Définition d'un triplet

Un triplet $t = (X_1, X_2, j)$ est défini par :

- Un premier sous-ensemble de jobs X_1 (représentés par leur ID)
- Un second sous-ensemble de jobs X_2 (représentés par leur ID)
- Un job j (représenté par son ID)

De plus, ces trois éléments doivent vérifier les propriétés suivantes :

- $X_1 \cap X_2 = \{j\}$.
- Il existe (au moins) une séquence ordonnant les jobs de $X_1 \cup X_2$, telle que :
  1. Les jobs dans $X_1 \backslash \{j\}$ sont traités *avant* j (sans interruption sur la machine 1).
  2. Les jobs dans $X_2 \backslash \{j\}$ sont traités *après* j, sans interruption sur la machine 2.

Autrement dit, un triplet représente la déviation d'un **chemin critique** de la machine 1 à la machine 2, qui influe sur le calcul du C_{max} d'une séquence partielle.

Note Par la suite, l'ensemble des séquences qui vérifient le deuxième point sera noté $\Sigma(t)$.

Propriétés d'un triplet

Soit t un triplet. Les dates de fin sur les machines 1 et 2 sont les mêmes pour toutes les séquences partielles appartenant à $\Sigma(t)$. C'est pourquoi on peut définir ces deux dates en tant que propriétés portées par le triplet t :

$$C_1(t) = \sum_{j \in X_1 \cup X_2} p_{1\,j}$$

$$C_2(t) = \sum_{j \in X_1} p_{1\,j} + \sum_{j \in X_2} p_{2\,j}$$

Ces deux dates vérifient la propriété suivante :

$$\forall \sigma \in \Sigma(t) \qquad \text{et} \begin{cases} C_1(\sigma) = C_1(t) \\ C_2(\sigma) = C_2(t) \end{cases}$$

Priorité associée à un triplet

Un ensemble de couples (t, σ_t) va être stocké en mémoire :

- t est un triplet ;
- σ_t est l'une des séquences partielles appartenant à $\Sigma(t)$.

À partir d'un triplet vide (c'est-à-dire tel que $X_1 = X_2 = \emptyset$, et j est indéfini), auquel est associé la séquence partielle vide ε, l'idée est de générer de nouveaux triplets, jusqu'à obtenir un triplet tel que $X_1 \cup X_2$ contient tous les jobs de l'instance : la séquence associée à ce triplet sera une séquence optimale.

Une valeur de priorité est utilisée pour choisir le triplet servant de base à la génération de nouveaux triplets. La valeur de la priorité d'un triplet t, notée $\mu(t)$, est définie à partir de la séquence partielle σ_t associée au triplet :

$$\mu(t) = C_3(\sigma_t)$$

Au cours de l'algorithme, la séquence σ_t associée au triplet t est susceptible d'être modifiée : cela implique que la priorité d'un triplet est une valeur *dynamique*.

À chaque itération, le triplet de priorité minimale est choisi. Les nouveaux triplets sont obtenus en ajoutant un job manquant après les jobs appartenant à $X_1 \cup X_2$. Deux cas sont à distinguer :

1. Le nouveau job ne dévie pas le chemin critique précédent.
2. Le nouveau job dévie le chemin critique précédent.

Les séquences partielles associées sont générées en ajoutant le même job manquant à la fin.

Le pseudo-code correspondant est indiqué ci-après.

1. $t_{init} \leftarrow (\emptyset, \emptyset, \bullet)$ /* *triplet vide ; le point signifie « intersection indéfinie »* */
2. $L \leftarrow \{t_{init}\}$ /* *permet de stocker et de trier tous les triplets* t */
3. $\sigma[t_{init}] = \varepsilon$ /* *associe à tout triplet* t *une séquence partielle* $\sigma[t] = \sigma_t$ */
4. **Tant que** $L \neq \emptyset$ **Faire**
5. Extraire $t^* = (X_1^*, X_2^*, j^*)$ de L, tel que $\mu(t^*) = \min_{t \in L}\{\mu(t)\}$ /* $= \min_{t \in L}\{C_3(\sigma[t])\}$ */
6. **Si** $X_1^* \cup X_2^*$ contient tous les jobs de l'instance **Alors**
7. Quitter la boucle **Tant que** /* σ_{t^*} *est une solution optimale* */
8. **Sinon**
9. **Pour tout** $j \notin X_1^* \cup X_2^*$ **Faire**
10. **Si** $C_1(t^*) + p_{1\,j} \leqslant C_2(t^*)$ **Alors**
11. $t \leftarrow (X_1^*, X_2^* \cup \{j\}, j^*)$ /* *le job* j *ne dévie pas le chemin critique* */
12. **Sinon**
13. $t \leftarrow (X_1^* \cup X_2^* \cup \{j\}, \{j\}, j)$ /* *le job* j *dévie le chemin critique* */
14. **Fin Si**
15. $\tau \leftarrow \sigma[t^*]|\bar{j}$ /* *ajout de* j *à la fin de* σ_{t^*} */
16. **Si** $t \notin L$ **Alors**
17. $L \leftarrow L \cup \{t\}$ /* *première insertion de* t *dans* L */
18. $\sigma[t] \leftarrow \tau$ /* *initialisation de* σ_t*, donc de* $\mu(t)$ */
19. **Sinon Si** $C_3(\tau) < \mu(t)$ **Alors**
20. $\sigma[t] \leftarrow \tau$ /* *mise à jour de* σ_t*, donc de* $\mu(t)$ *qui ne peut que diminuer* */
21. **Fin Si**
22. **Fin Pour**
23. **Fin Si**
24. **Fin Tant que**
25. **retourner** $\sigma[t^*]$

Algorithme 1: Pseudo-code de l'algorithme à base de triplets

Des schémas illustrant les deux situations pour la génération du nouveau triplet t à partir du triplet $t^* = (X_1^*, X_2^*, j^*)$ par ajout du job j sont proposés page 32.

4.4.2 Complexité

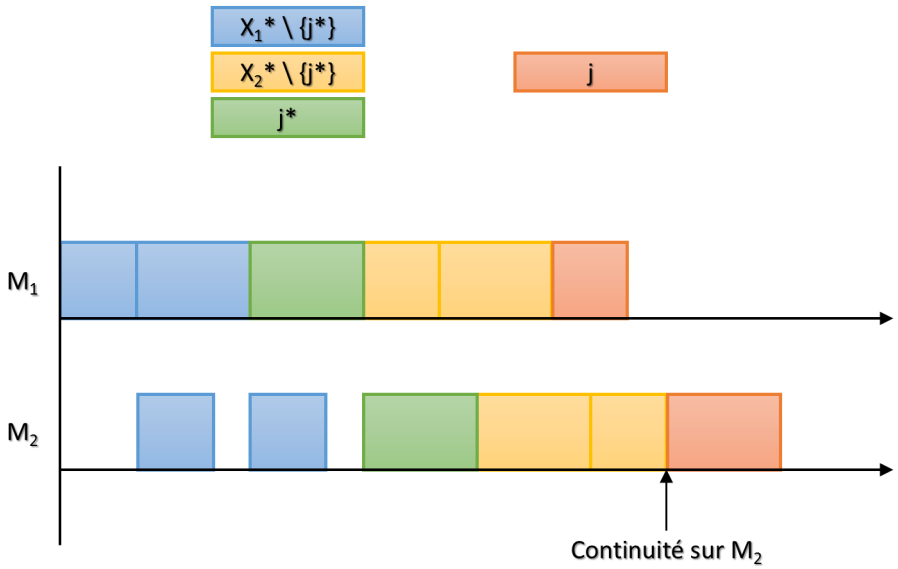
Soit une instance à n jobs. Notons K_n le nombre de triplets considérés au cours de l'algorithme.

En utilisant une ou plusieurs structures adaptées pour ordonner l'ensemble des couples (t, σ_t) selon différents critères de tri, les complexités relatives aux opérations « élémentaires » réalisées sur l'ensemble des triplets peuvent être bornées ainsi :

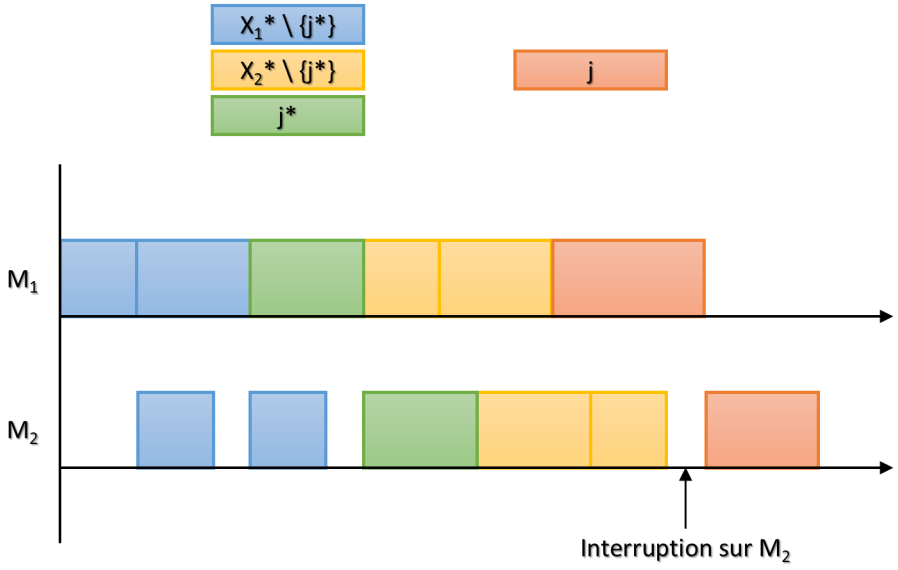
- La recherche, l'insertion ou la suppression d'un triplet en particulier est au pire cas en $\mathcal{O}(n \ln(K_n))$ (la multiplication par n correspond à la complexité de la comparaison élémentaire entre deux triplets, par exemple de manière lexicographique, donc au pire cas en $\mathcal{O}(n)$).
- La recherche, l'insertion ou la suppression du triplet le plus prioritaire est au pire cas en $\mathcal{O}(\ln(K_n))$ (la comparaison de la priorité entre deux triplets est réalisable en temps constant).

Pour une itération de la boucle **Tant que**, la boucle **Pour tout** est exécutée au plus n fois. Puisque la boucle **Tant que** est réalisée au plus K_n fois, la complexité globale de l'algorithme est donc, au pire cas, en $\mathcal{O}(K_n \times n \times n \ln(K_n)) = \mathcal{O}^*(K_n \ln(K_n))$.

Par ailleurs, le nombre K_n de triplets est inférieur à $n3^n$. En effet, étant donné un triplet $t = (X_1, X_2, j)$:



Cas où le chemin critique n'est pas dévié : $C_1(t^*) + p_{1\,j} \leqslant C_2(t^*)$



Cas où le chemin critique est dévié : $C_1(t^*) + p_{1\,j} > C_2(t^*)$

FIGURE 4.3 – Génération d'un nouveau triplet t par ajout d'un job j à un triplet $t^* = (X_1^*, X_2^*, j^*)$

- Pour tout job j' de l'instance : soit j' appartient à X_1, soit j' appartient à $X_2 \backslash \{j\}$, soit j' n'appartient ni à X_1 ni à X_2 (trois possibilités).
- j peut prendre au plus $|X_1 \cup X_2| \leqslant n$ valeurs.

On en déduit que la complexité globale de l'algorithme est en $\mathcal{O}^*(n3^n \ln(n3^n)) = \mathcal{O}^*(3^n)$.

4.4.3 Implémentation

Codage d'un triplet

Il a fallu choisir une structure pour représenter les sous-ensembles X_1 et X_2 d'un triplet t, qui permette de réaliser rapidement la fusion des deux (génération d'un nouveau triplet, lorsque le chemin critique est dévié). Utiliser un conteneur qui trie ses éléments (ici : des ID de jobs) semble donc adapté : ainsi, la première possibilité envisagée a été l'utilisation d'un `std::set<unsigned int>`. Cependant, ces derniers sont souvent implémentés sous la forme d'arbres de recherche : l'espace mémoire correspondant est d'assez grande taille. C'est pourquoi un autre conteneur a finalement été utilisé : une `std::list<unsigned int>` (liste chaînée), maintenue triée dans le code. Ce conteneur dispose d'ailleurs d'une méthode prédéfinie `merge()` capable de fusionner des listes triées au préalable. Ainsi, X_1 et X_2 occupent un espace mémoire de taille réduite ; de plus, leur fusion est réalisée en $\mathcal{O}(|X_1| + |X_2|)$ (complexité temporelle minimale).

Stockage des couples (t, σ_t) permettant plusieurs types d'accès aux triplets (recherche d'un triplet en particulier, recherche du triplet le plus prioritaire)

L'implémentation d'un tel conteneur est une tâche complexe à réaliser. Plusieurs pistes de conception d'une structure de données adéquate ont été explorées, mais les contraintes techniques sous-jacentes étaient toujours très forte, notamment en ce qui concerne la gestion des priorités *dynamiques* (par exemple, l'utilisation d'une file de priorité « standard » telle que `std::priority_queue` ne permet que la manipulation d'éléments avec une priorité *statique*).

Après une phase de recherche d'un conteneur proposant différentes manières d'accéder à ses éléments, l'un d'entre eux a été retenu : `boost::multi_index_container`, disponible dans la librairie C++ Boost.

- Le stockage des données est totalement transparent pour l'utilisateur.
- Les données sont accessibles *via* des interfaces spécifiques, appelées « index » : ce sont des vues, qui trient les éléments selon un critère défini au moment de la déclaration du conteneur. Une vue est aussi performante qu'un `std::set` pour retrouver un élément.

Le modèle de classe `boost::multi_index_container` a été instancié ainsi (voir schéma page 36) :

- Les éléments stockés sont représentés par une classe `Element` qui comporte deux attributs : un triplet t et la séquence associée σ_t
- Deux critères de tri sont utilisés :
  1. comparaison lexicographique entre triplets t, pour retrouver un triplet en particulier
  2. tri par priorité $\mu(t) = C_{max}(\sigma_t)$, pour retrouver le triplet le plus prioritaire

4.5 Algorithme à base de « triplets », avec élimination de solutions dominées

4.5.1 Présentation

Cet algorithme reprend l'algorithme à base de triplets, auquel est ajoutée la condition de dominance entre séquences partielles utilisée par l'algorithme de programmation dynamique.

Intégration de la condition de dominance entre séquences partielles, pour minimiser le nombre de triplets à étudier

Soit S un sous-ensemble de jobs. Soit $T(S)$ l'ensemble des triplets $t = (X_1, X_2, j)$ tels que $X_1 \cup X_2 = S$. Soit $\Sigma(S)$ l'ensemble des séquences partielles σ_t associées à ces triplets : $\Sigma(S) = \{\sigma_t \ / \ t \in T(S)\}$. Il s'agit d'un ensemble de séquences partielles qui ordonnent les mêmes jobs (ceux qui appartiennent à S). La condition de dominance peut donc être appliquée entre deux éléments de $\Sigma(S)$: les séquences dominées, n'appartenant pas au front de Pareto, peuvent être ignorées pour la recherche d'une solution optimale.

On définit alors la relation de dominance entre deux triplets $t = (X_1, X_2, j)$ et $t' = (X'_1, X'_2, j')$ à partir de la relation de dominance entre séquences partielles :

$$t \preccurlyeq t' \quad \Leftrightarrow \quad \text{et} \begin{cases} X_1 \cup X_2 = X'_1 \cup X'_2 \\ \sigma_t \preccurlyeq \sigma_{t'} \end{cases}$$

Pour rechercher une solution optimale, il est suffisant de prendre en compte seulement les triplets qui ne sont pas dominés.

Voici le pseudo-code de l'algorithme correspondant :

1. $t_{init} \leftarrow (\emptyset, \emptyset, \bullet)$ /* *triplet vide ; le point signifie « intersection indéfinie »* */
2. $L \leftarrow \{t_{init}\}$ /* *permet de stocker et de trier tous les triplets t* */
3. $\sigma[t_{init}] = \varepsilon$ /* *associe à tout triplet t une séquence partielle* $\sigma[t] = \sigma_t$ */
4. **Tant que** $L \neq \emptyset$ **Faire**
5. Extraire $t^* = (X_1^*, X_2^*, j^*)$ de L, tel que $\mu(t^*) = \min_{t \in L}\{\mu(t)\}$ /* $= \min_{t \in L}\{C_3(\sigma[t])\}$ */
6. **Si** $X_1^* \cup X_2^*$ contient tous les jobs de l'instance **Alors**
7. Quitter la boucle **Tant que** /* σ_{t^*} *est une solution optimale* */
8. **Sinon**
9. **Pour tout** $j \notin X_1^* \cup X_2^*$ **Faire**
10. **Si** $C_1(t^*) + p_{1\,j} \leqslant C_2(t^*)$ **Alors**
11. $t \leftarrow (X_1^*, X_2^* \cup \{j\}, j^*)$ /* *le job j ne dévie pas le chemin critique* */
12. **Sinon**
13. $t \leftarrow (X_1^* \cup X_2^* \cup \{j\}, \{j\}, j)$ /* *le job j dévie le chemin critique* */
14. **Fin Si**
15. $\tau \leftarrow \sigma[t^*]|\bar{j}$ /* *ajout de j à la fin de* σ_{t^*} */
16. **Si** $t \notin L$ **Alors**
17. **Si** t n'est dominé par aucun triplet de L /* $\sigma_t = \tau$ */ **Alors**
18. $L \leftarrow L \cup \{t\}$ /* *première insertion de t dans L* */
19. $\sigma[t] \leftarrow \tau$ /* *initialisation de* σ_t*, donc de* $\mu(t)$ */
20. Retirer de L tous les triplets dominés par t
21. **Fin Si**
22. **Sinon Si** $C_3(\tau) < \mu(t)$ **Alors**
23. $\sigma[t] \leftarrow \tau$ /* *mise à jour de* σ_t*, donc de* $\mu(t)$ *qui ne peut que diminuer* */
24. Retirer de L tous les triplets dominés par t
25. **Fin Si**
26. **Fin Pour**
27. **Fin Si**
28. **Fin Tant que**
29. **retourner** $\sigma[t^*]$

Algorithme 2: Pseudo-code de l'algorithme à base de triplets, avec élimination de solutions dominées

4.5.2 Complexité

Les complexités relatives aux opérations « élémentaires » réalisées sur l'ensemble des couples (t, σ_t) restent inchangées.

L'ensemble des classes d'équivalence regroupant les triplets dont les unions des deux sous-ensembles sont égales à un même sous-ensemble de jobs contient 2^n éléments (quantité égale au nombre de parties d'un ensemble de taille n). Ainsi, en ajoutant une structure permettant de grouper les couples par classe, la recherche des triplets qui peuvent potentiellement dominer un triplet donné (dont la classe est connue) peut donc être réalisée en $\mathcal{O}(\ln(2^n)) = \mathcal{O}(n)$. De plus, si l'ensemble des séquences formant le front de Pareto d'une classe est trié, la recherche de triplets qui dominent ou qui sont dominés par un triplet donné peut également être effectuée en $\mathcal{O}(\ln(2^n)) = \mathcal{O}(n)$ (car tout front de Pareto contient toujours moins de 2^n points).

En utilisant la notation $\mathcal{O}^*(\alpha^n)$, la complexité temporelle globale de cette variante de l'algorithme à base de triplets est donc identique à celle de l'algorithme de départ : $\mathcal{O}^*(3^n)$. Néanmoins, cela nécessite une augmentation importante de l'espace mémoire occupé.

4.5.3 Implémentation

La même structure de données utilisée par l'algorithme à base de triplets dans sa première version a été réutilisée, en ajoutant une nouvelle vue qui se comporte comme un `std::multiset` (voir schéma page 36) : deux triplets (*a priori* distincts) y sont considérés comme égaux lorsque les unions des deux sous-ensembles de jobs X_1 et X_2 sont égales, c'est-à-dire, si les deux triplets appartiennent à la même classe d'équivalence. L'appel à la méthode `equal_range()` permet d'obtenir l'ensemble des triplets d'une classe d'équivalence. À chaque ajout ou modification d'un couple (t, σ_t) dans le conteneur, la condition de dominance est prise en compte, pour ne conserver que le front de Pareto.

Cette implémentation n'est pas optimale : pour certains traitements, la complexité réelle est supérieure à la complexité minimale pouvant être théoriquement atteinte.

1. La méthode `equal_range()` ne procède pas en $\mathcal{O}(\ln(2^n))$ mais en $\mathcal{O}(\ln(n3^n))$. Les couples (t, σ_t) sont bien regroupés par classes d'équivalence (les uns à la suite des autres dans la nouvelle vue) ; néanmoins, la méthode ne recherche pas une classe parmi d'autres, mais le premier et le dernier triplet d'une classe parmi l'ensemble de tous les triplets préalablement regroupés par classes. Cependant, même si la complexité est légèrement supérieure, elle reste en $\mathcal{O}(n)$.
2. La recherche des séquences qui dominent ou qui sont dominées par une autre séquence se fait en parcourant un à un les couples (t, σ_t) de la classe d'équivalence retournée par la méthode `equal_range()` : autrement dit, le traitement a une complexité linéaire (en fonction de la taille de la classe), alors que la complexité pourrait être logarithmique si les séquences étaient ordonnées.

Remarque Le développement en intégralité d'un conteneur prenant non seulement en compte toutes les contraintes de l'algorithme à base de triplets initial, mais aussi la relation de dominance utilisée pour ne conserver que des fronts de Pareto, est extrêmement complexe. C'est pourquoi cette implémentation, certes non optimale, mais ayant le mérite de prendre tous ces critères en compte, a tout de même été étudiée.

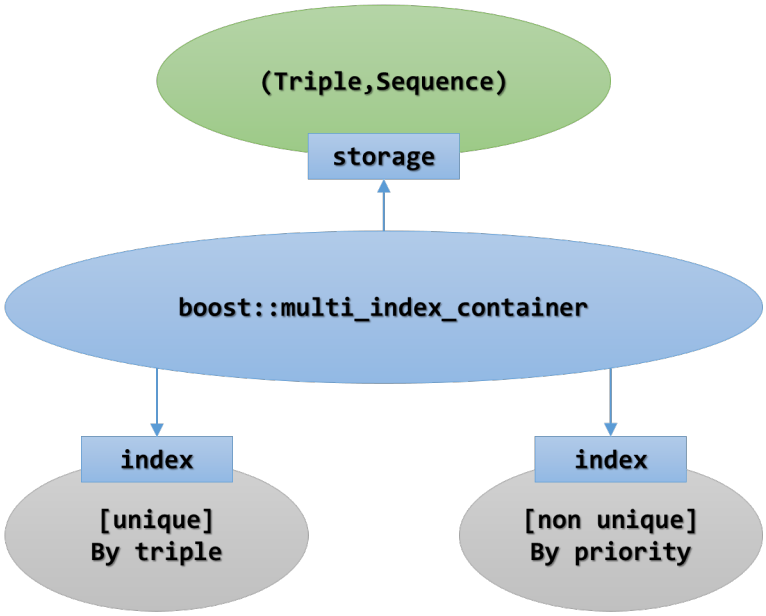


FIGURE 4.4 – Algorithme à base de triplets : structure de données

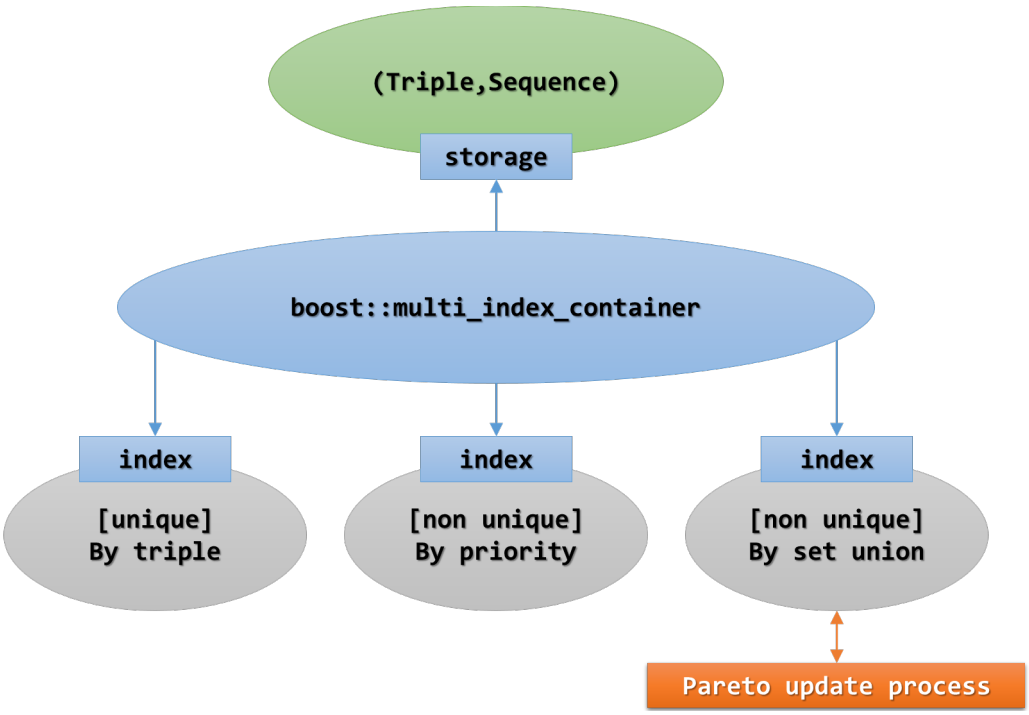


FIGURE 4.5 – Algorithme à base de triplets incluant une relation de dominance : structure de données

4.6 Algorithme de type *Branch & Bound* (référence)

4.6.1 Présentation

Cet algorithme exact a été proposé par **Ladhari** et **Haouari**. Il trouve des solutions optimales pour le problème $F|prmu|C_{max}$ (nombre de machines quelconque, restriction aux solutions de permutation). Ces performances sont connues dans le domaine de la recherche opérationnelle : les temps d'exécution sont extrêmement rapides, dans le cas où le nombre de machines est suffisamment petit, ce qui est le cas pour le $F_3||C_{max}$.

Cet algorithme a été étudié pour deux raisons principales :

1. Étude théorique d'un algorithme de type *Branch & Bound* (tous les autres algorithmes étudiés en relation avec le $F_3||C_{max}$ se rapprochent plutôt de la programmation dynamique)
2. Mesure des temps d'exécution sur la même machine sur un même jeu d'instances, pour avoir une base de comparaison commune pour analyser les performances des différents algorithmes

Structure arborescente

Les nœuds de l'arbre de recherche construit et parcouru dans l'algorithme sont définis ainsi :

1. Une séquence partielle ordonnée de jobs σ_1, de longueur N_1, liste les jobs à traiter en premier
2. Une séquence partielle ordonnée de jobs σ_2, de longueur N_2, liste les jobs à traiter en dernier

La profondeur d'un nœud est égale au nombre de jobs placés : $N_1 + N_2$

Les fils d'un nœud sont construits en plaçant un nouveau job au milieu des deux séquences, de deux manières différentes :

1. Soit en ajoutant un job manquant à la fin de σ_1
2. Soit en ajoutant un job manquant au début de σ_2

Au niveau de la racine, aucun job n'est placé : $\sigma_1 = \sigma_2 = \varepsilon$ $(N_1 = N_2 = 0)$. Au niveau des feuilles, tous les jobs sont placés : $N_1 + N_2$ est égal au nombre de jobs dans l'instance.

Le schéma page 39 représente les informations associées à un nœud.

Bornes inférieures basées sur la relaxation de contraintes sur plusieurs machines

Tout algorithme *Branch & Bound* est défini notamment par l'utilisation de bornes inférieures au niveau de chaque nœud. Ces bornes inférieures peuvent permettre d'ignorer certaines branches de l'arbre de recherche.

Ici, le principe retenu est d'ôter la contrainte suivante :

Chaque machine ne peut traiter qu'un seul job à la fois.

Dans l'algorithme initialement proposé par **Ladhari** et **Haouari**, cette contrainte est relâchée :

1. Soit pour toutes les machines, sauf une : étude du problème $1|r_j, q_j|C_{max}$
   (1 machine ; dates de disponibilité r_j, durées de livraison q_j ; minimiser le C_{max})
2. Soit pour toutes les machines, sauf deux : étude du problème $F2|r_j, \ell_j, q_j, prmu|C_{max}$
   (2 machines ; dates de disponibilité r_j, temps de latence minimal ℓ_j entre la date de fin sur la machine 1 et la date de début sur la machine 2, durées de livraison q_j, même ordre de passage des jobs sur les deux machines ; minimiser le C_{max})

Les schémas de la page 39 illustrent comment effectuer la conversion du problème initial ($F|prmu|C_{max}$) vers l'un ou l'autre des deux problèmes relâchés (pour un job).

Selon la profondeur du nœud, un type de relaxation est choisi. Une borne inférieure ou une solution exacte est calculée pour le nouveau problème obtenu (au total, 7 méthodes sont proposées dans l'article décrivant l'algorithme, dont 3 sont utilisées lors de l'implémentation). Cette valeur constitue une borne inférieure pour le problème initial.

4.6.2 Complexité

La complexité au pire cas d'un algorithme *Branch & Bound* est toujours en $\mathcal{O}^*(n!)$: si les bornes inférieures ne permettent pas d'éliminer des branches dans l'arbre de recherche, l'espace des solutions est exploré entièrement. Cependant, les bornes inférieures utilisées dans cet algorithme sont, en pratique, extrêmement efficaces.

Comme évoqué précédemment, certaines bornes inférieures du problème initial sont des bornes inférieures d'un problème relâché. Ce genre de bornes inférieures est utilisé pour des nœuds de faible profondeur, dans le but d'élaguer rapidement certaines branches le plus tôt possible dans l'algorithme. La complexité de ce type de bornes inférieures est toujours pseudo-polynomial en fonction de m et n : autrement dit, le temps de calcul pour obtenir ces bornes est court, la valeur obtenue étant de qualité moyenne voire basse.

D'autres bornes inférieures du problème initial sont obtenues en calculant des solutions exactes des problèmes relâchés. Ce genre de bornes inférieures est utilisé pour des nœuds profonds, dans le but d'obtenir une valeur aussi grande que possible (haute qualité). La complexité de ce type de bornes inférieures est élevée : en effet, d'autres algorithmes de type *Branch & Bound* sont utilisés sur les problèmes relâchés. Toutefois, puisque les nœuds sont profonds, il reste peu de jobs à placer. Or, plus la taille des entrées est petite, plus le temps d'exécution est court. Le temps nécessaire au calcul de ce genre de bornes inférieures devrait donc être raisonnable.

4.6.3 Adaptation du code original

Le code C de l'algorithme a pu être récupéré auprès de ses auteurs ; plus exactement, ceux-ci ont transmis un code très proche, ayant servi dans le cadre d'un travail de thèse qu'ils ont encadré.

Dans un premier temps, le travail a donc consisté à comprendre la structure de ce code (un fichier source unique d'environ 1500 lignes). Cette étape a permis d'identifier certaines limitations. L'initialisation de certaines variables était réalisée « en dur » dans le code, seulement pour certaines valeurs du nombre de machines (5, 10 ou 20) ; la valeur appropriée n'a malheureusement pas pu être déduite pour le cas de 3 machines. Pour contourner le problème, une seule borne inférieure a été finalement utilisée pour tous les nœuds (au lieu de trois initialement, l'une d'entre elles étant sélectionnée en fonction de la profondeur du nœud courant) : il s'agit d'une borne inférieure basée sur une relaxation de contraintes pour toutes les machines sauf une.

Dans un second temps, la fonction `main()` a été renommée, son prototype et son code ont été transformés, pour qu'elle puisse lire les données et écrire les résultats dans les classes C++ prévues à cet effet, déjà implémentées pour les algorithmes présentés précédemment. Cela a nécessité l'utilisation de fonctions « *wrapper* » (mécanisme permettant de définir des fonctions dont le corps contient des instructions C++ mais pouvant être appelées dans un code C).

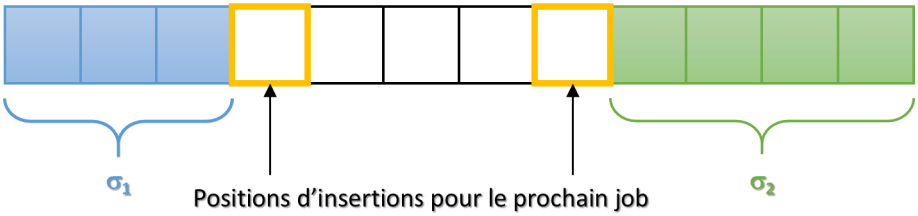
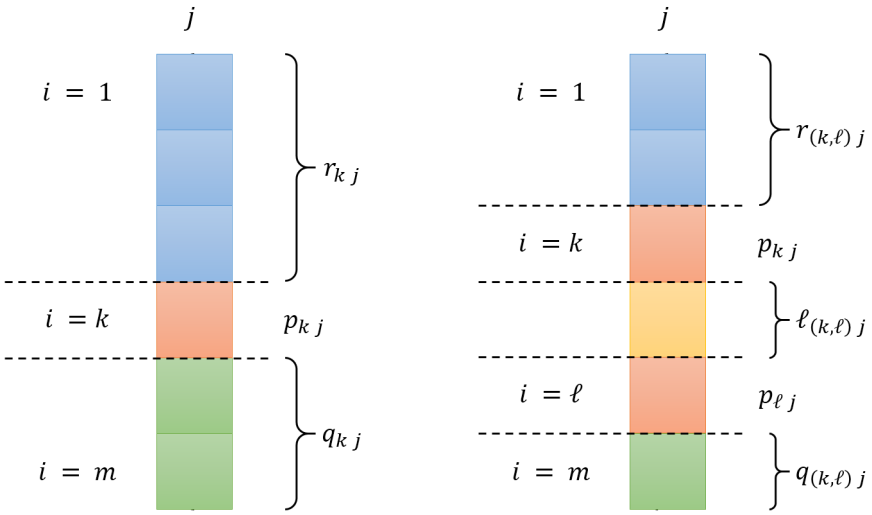


FIGURE 4.6 – Algorithme *Branch & Bound* : information associée à un nœud



$1|r_j, q_j|C_{max}$
(machine k)

$F2|r_j, \ell_j, q_j, prmu|C_{max}$
(machines k et ℓ)

FIGURE 4.7 – Deux relâchements possibles du problème $F|prmu|C_{max}$

4.7 Comparaison des performances des algorithmes

4.7.1 Conditions de réalisation des tests de performance

Génération d'un jeu d'instances

Afin de comparer les performances des algorithmes implémentés, il a fallu générer un jeu d'instances, pour mesurer le temps d'exécution des différents programmes (correspondant aux différents algorithmes) sur les mêmes instances.

Pour ce faire, un programme auxiliaire a été réalisé en C++. En entrée, l'utilisateur indique une quantité d'instances à générer, ainsi que le nombre de jobs et le nombre de machines (qui seront les mêmes pour toutes les instances générées). En sortie, l'utilisateur obtient un fichier texte contenant les instances générées, dans un format pouvant être lu directement par les programmes de résolution (ce format est similaire à celui utilisé pour les instances sur le site Taillard).

Chaque durée de traitement ($p_{i\,j}$) est obtenue en tirant aléatoirement un entier entre 1 et 100, selon une distribution uniforme. Pour y parvenir, le programme utilise les fonctionnalités suivantes de la STL, fournies dans l'entête `<random>` :

- `std::default_random_engine` (moteur pour la génération de nombres aléatoires, initialisé à partir de la date à laquelle le programme est exécuté)
- `std::uniform_int_distribution` (modification des valeurs générées par le moteur, pour simuler une loi de distribution uniforme sur un ensemble de valeurs discret)

Au total, 180 instances ont été générées : cela correspond à 20 instances pour chaque taille, de 5 à 50 jobs par pas de 5.

Machine utilisée pour la réalisation des tests

Voici les caractéristiques de la machine sur laquelle les tests ont été exécutés :

- Type : Ordinateur portable
- Constructeur : Samsung
- Modèle : NP300E5C-AF2FR
- RAM : 4 Go
- Processeur : Intel Celeron CPU B820, 1,70 GHz
- Système d'exploitation : Windows 7 64 bits

Remarque Il s'agit de l'ordinateur proposé aux étudiants par la Région Centre en 2012-2013, dans le cadre de l'opération « Ordi Centre ».

Les applications ont été compilées sur cette machine, avec Visual C++ 11.0 (Visual Studio 2012).

4.7.2 Temps de calcul moyens, minimaux et maximaux

Les temps d'exécution mesurés pour chaque algorithme sur chaque instance, ainsi que les temps d'exécution moyens, minimaux et maximaux pour chaque taille d'instance, sont disponibles aux pages suivantes du rapport :

- Algorithme de programmation dynamique, avec élimination de solutions dominées :

1. Implémentation utilisant `Pareto::Extractor` : page 44
2. Implémentation utilisant `Pareto::Set` : page 44

- Algorithme à base de triplets :
  1. Version de base : page 46
  2. Version incluant une condition de dominance : page 46
- Algorithme *Branch & Bound* : page 47

Une synthèse est également proposée page 48.

Remarque Les tableaux répertorient les temps d'exécution relatifs aux instances pour lesquelles les algorithmes se sont exécutés jusqu'à leur terme. Lorsqu'une taille d'instance est manquante, cela signifie que le programme s'est arrêté prématurément. En l'occurrence, il se trouve que la raison de cet arrêt prématuré est la même pour tous les algorithmes concernés : une exception `std::bad_alloc` est levée (allocation mémoire impossible, car la taille de l'espace occupé par le programme devient trop grande).

4.7.3 Analyse des résultats expérimentaux

Comparaison des solutions optimales retournées par les différents algorithmes

Pour une instance donnée, les algorithmes (notamment le *Branch & Bound*, qui utilise un code extérieur déjà testé auparavant) retournent des séquences optimales qui peuvent différer, mais pour lesquelles le C_{max} est identique. Cela démontre que les autres algorithmes étudiés fournissent eux aussi des solutions exactes, et que l'implémentation est bien conforme au pseudo-code.

Les valeurs des solutions optimales sont répertoriées dans le tableau page 47.

Analyse statistique détaillée pour l'implémentation utilisant le conteneur `Pareto::Set` de l'algorithme de programmation dynamique avec élimination de solutions dominées

Pour chaque `Pareto::Set` (un conteneur/front de Pareto associé à l'un des 2^n sous-ensembles de jobs), deux taux ont été définis.

Taux de conservation

Soit N le nombre de tentatives successives d'insertion de points dans un front de Pareto. À l'issue de ces N tentatives d'insertion :

1. Les points dominés ne font pas partie du front (quantité : N_d).
2. Les points non dominés font partie du front (quantité : $N_{nd} = N - N_d$).

On définit le taux de conservation par :

$$\text{Taux de conservation} = \frac{N_{nd}}{N}$$

Remarque Le taux de conservation est *indépendant* de l'ordre dans lequel les N tentatives d'insertion de points dans le front de Pareto sont réalisées.

Taux d'insertion

Soit N le nombre de tentatives successives d'insertion de points dans un front de Pareto. Parmi les N points, au moment de la tentative d'insertion :

1. Certains sont dominés par ceux déjà présents dans le front (quantité : N_i) : dans ce cas, le front de Pareto n'est pas modifié.
2. Certains ne sont dominés par aucun autre déjà présent dans le front (quantité : $N_{ni} = N - N_i$) : dans ce cas, le front de Pareto est mis à jour (ajout du nouveau point, suppression éventuelle des points qu'il domine).

On définit le taux d'insertion par :

$$\text{Taux d'insertion} = \frac{N_{ni}}{N}$$

Remarque Le taux d'insertion est *dépendant* de l'ordre dans lequel les N tentatives d'insertion de points dans le front de Pareto sont réalisées.

Mesures des taux de conservation et d'insertion

Les moyennes par instance des taux d'insertion relevés sur chaque `Pareto::Set`, ainsi que les moyennes par taille d'instance, sont disponibles page 45.

Analyse des taux de conservation et d'insertion

On peut constater que le taux de conservation diminue avec la taille de l'instance. Cela signifie que la proportion de séquences partielles éliminées augmente : autrement dit, le caractère discriminant de la condition de dominance augmente lorsque le nombre de jobs augmente.

Le taux d'insertion suit la même tendance. Cela signifie que la proportion de mise à jour des fronts de Pareto à chaque tentative d'insertion diminue elle aussi.

Comparaison des performances des différents algorithmes

L'algorithme *Branch & Bound* est nettement plus performant que tous les autres algorithmes. D'une part, il est extrêmement rapide ; d'autre part, il est capable de résoudre des instances de très grande taille. Avec moins de 50 jobs, les durées mesurées sont quasiment nulles. Des tests complémentaires ont été réalisés sur la même machine avec des instances à 500 jobs : le temps d'exécution est de l'ordre de 5 secondes. Cet algorithme est donc extrêmement efficace pour résoudre le problème du $F_3||C_{max}$.

Les performances des autres algorithmes sont très nettement inférieures :

1. Ils ne parviennent pas à trouver des solutions à partir d'une taille d'instance assez petite : aucun d'entre eux ne passe la barre des 25 jobs.
2. Les temps d'exécution sont très longs, comparés à ceux du *Branch & Bound*.

Il existe différents éléments de réponse susceptibles d'expliquer ces résultats :

- L'algorithme *Branch & Bound* est basé sur un parcours en profondeur d'abord d'un arbre de recherche. Les implémentations des autres algorithmes, qui progressent par phase, réalisent plutôt un parcours en largeur d'abord de l'espace des séquences de jobs partielles : par conséquent, l'espace mémoire requis est plus important (complexité spatiale exponentielle), avec un goulot d'étranglement correspondant à la phase pour laquelle la taille des séquences partielles est égale à la moitié du nombre de jobs. Ainsi, il pourrait être intéressant de s'intéresser à une implémentation en profondeur d'abord (complexité spatiale polynomiale) de ces autres algorithmes ; toutefois, ceci se ferait au prix d'une augmentation significative de la complexité temporelle (par exemple, pour l'algorithme à base de triplets dans sa version initiale, la complexité temporelle passerait de $\mathcal{O}^*(3^n)$ à $\mathcal{O}^*(9^n)$).
- L'algorithme *Branch & Bound* a été implémenté en langage C, alors que les autres algorithmes ont été implémentés en langage C++. La sur-couche « objet » de ce deuxième langage (appels aux constructeurs et destructeurs, passage par des méthodes publiques pour accéder de manière sécurisée et transparente aux attributs privés, etc) engendre un sur-coût temporel. Par exemple, en rendant certaines méthodes `inline` (appel à une méthode remplacé par le corps de celle-ci lors de la compilation), les temps d'exécution pourraient éventuellement être réduits.
- Le code de l'algorithme *Branch & Bound* comprend la définition de structures de données spécifiquement conçues pour le problème étudié. L'implémentation des autres algorithmes repose sur l'utilisation de conteneurs génériques (modèles de classes de la STL ou de la librairie Boost) : ceux-ci ont été choisis car ils minimisent autant que possible la complexité des traitements d'accès aux données dont les différents algorithmes ont besoin. Cependant, il pourrait être envisageable de remplacer ces conteneurs par d'autres structures de données définies « manuellement » (cette tâche pouvant s'avérer être très complexe), en particulier pour l'algorithme à base de triplets incluant une condition de dominance.

Data ID	5 jobs	10 jobs	15 jobs	20 jobs
0	0	0,031	2,028	79,653
1	0	0,031	1,497	80,059
2	0	0,031	1,825	149,9
3	0	0,031	2,652	68,764
4	0	0,015	2,34	70,2
5	0	0,031	1,731	83,35
6	0	0,031	1,653	65,769
7	0	0,046	1,326	113,739
8	0	0,015	1,56	67,158
9	0	0,015	1,357	91,852
10	0	0,031	1,965	85,129
11	0	0,031	2,028	102,523
12	0	0,015	2,293	60,559
13	0	0,031	1,341	86,424
14	0	0,015	4,492	85,8
15	0	0,031	1,591	78,468
16	0	0,031	1,56	85,628
17	0	0,031	2,293	73,725
18	0	0,031	1,107	92,695
19	0	0,031	1,809	118,731
Min time (s)	**0,000**	**0,015**	**1,107**	**60,559**
Max time (s)	**0,000**	**0,046**	**4,492**	**149,900**
Mean time (s)	**0,000**	**0,028**	**1,922**	**87,006**
Min time (mn)	**0,000**	**0,000**	**0,018**	**1,009**
Max time (mn)	**0,000**	**0,001**	**0,075**	**2,498**
Mean time (mn)	**0,000**	**0,000**	**0,032**	**1,450**

TABLE 4.1 – Résultats pour l'algorithme de programmation dynamique (implémentation 1)

Data ID	5 jobs	10 jobs	15 jobs	20 jobs
0	0	0,031	1,201	51,682
1	0	0,015	0,92	49,639
2	0	0,031	1,138	82,524
3	0	0,015	1,606	45,193
4	0	0,015	1,31	48,859
5	0	0,015	1,076	53,071
6	0	0,015	1,045	44,241
7	0	0,031	0,889	65,504
8	0	0,015	1,06	45,832
9	0	0,015	0,904	54,912
10	0	0,031	1,138	53,788
11	0	0,015	1,216	62,758
12	0	0,015	1,419	42,354
13	0	0,015	0,904	54,194
14	0	0,015	2,823	53,835
15	0	0,015	1,045	48,562
16	0	0,015	0,951	53,679
17	0	0,031	1,388	46,456
18	0	0,015	0,826	57,205
19	0	0,031	1,138	67,47
Min time (s)	**0,000**	**0,015**	**0,826**	**42,354**
Max time (s)	**0,000**	**0,031**	**2,823**	**82,524**
Mean time (s)	**0,000**	**0,020**	**1,200**	**54,088**
Min time (mn)	**0,000**	**0,000**	**0,014**	**0,706**
Max time (mn)	**0,000**	**0,001**	**0,047**	**1,375**
Mean time (mn)	**0,000**	**0,000**	**0,020**	**0,901**

TABLE 4.2 – Résultats pour l'algorithme de programmation dynamique (implémentation 2)

Data ID	5 jobs	10 jobs	15 jobs	20 jobs
0	51,09	24,73	14,06	10,45
1	53,28	23,90	13,94	10,02
2	56,41	26,41	15,18	10,12
3	59,91	24,57	15,88	10,10
4	49,32	23,12	15,30	10,23
5	58,32	25,66	14,20	10,11
6	56,40	24,64	14,89	10,55
7	52,79	27,78	14,28	10,35
8	59,33	24,29	13,80	10,19
9	55,55	23,54	13,76	10,13
10	58,82	25,13	15,33	10,13
11	52,84	22,81	14,78	10,44
12	59,25	24,02	14,17	9,94
13	57,34	25,79	14,77	9,90
14	51,12	24,51	16,41	10,89
15	57,36	23,97	14,37	10,65
16	52,99	24,61	14,92	9,86
17	55,03	26,54	15,30	9,82
18	53,71	25,56	13,81	9,60
19	56,14	25,51	14,97	10,83
Min (%)	**49,32**	**22,81**	**13,76**	**9,60**
Max (%)	**59,91**	**27,78**	**16,41**	**10,89**
Mean (%)	**55,35**	**24,85**	**14,71**	**10,22**

Taux de conservation

Data ID	5 jobs	10 jobs	15 jobs	20 jobs
0	60,16	47,42	28,13	33,53
1	72,97	41,94	20,38	26,75
2	62,34	48,07	33,76	24,40
3	69,50	44,06	38,78	25,80
4	70,78	49,89	32,74	36,40
5	58,32	32,87	27,15	31,16
6	79,90	40,77	31,66	25,93
7	91,11	38,45	30,93	24,67
8	83,75	46,43	43,86	28,11
9	71,18	49,56	29,87	21,64
10	83,35	49,01	25,51	30,25
11	76,71	52,63	36,70	32,63
12	76,96	45,62	43,94	30,81
13	75,38	45,60	39,41	27,53
14	89,19	48,14	36,07	25,68
15	74,68	64,99	38,97	20,32
16	57,42	41,06	25,36	35,00
17	63,52	55,58	34,52	15,47
18	78,79	39,97	39,91	22,07
19	76,56	36,51	42,27	26,11
Min (%)	**57,42**	**32,87**	**20,38**	**15,47**
Max (%)	**91,11**	**64,99**	**43,94**	**36,40**
Mean (%)	**73,63**	**45,93**	**34,00**	**27,21**

Taux d'insertion

TABLE 4.3 – Autres mesures pour l'algorithme de programmation dynamique (implémentation 2)

Data ID	5 jobs	10 jobs
0	0	0,967
1	0	0,842
2	0	1,216
3	0	1,076
4	0	0,639
5	0,015	0,405
6	0	0,982
7	0	0,811
8	0	1,294
9	0	0,733
10	0	1,232
11	0	0,624
12	0	0,764
13	0	1,014
14	0	1,138
15	0	0,483
16	0	0,951
17	0	0,998
18	0	0,904
19	0	1,045
Min time (s)	**0,000**	**0,405**
Max time (s)	**0,015**	**1,294**
Mean time (s)	**0,001**	**0,906**
Min time (mn)	**0,000**	**0,007**
Max time (mn)	**0,000**	**0,022**
Mean time (mn)	**0,000**	**0,015**

TABLE 4.4 – Résultats pour l'algorithme à base de triplets

Data ID	5 jobs	10 jobs	15 jobs	20 jobs
0	0	0,062	16,473	745,884
1	0	0,078	6,973	1130,02
2	0	0,109	11,31	2663,28
3	0	0,062	16,239	819,609
4	0	0,046	10,124	1089,3
5	0	0,062	10,342	1128,07
6	0	0,093	7,488	602,005
7	0	0,156	9,063	1103,44
8	0	0,062	13,15	775,586
9	0	0,062	7,644	1318,51
10	0	0,078	10,358	1700,06
11	0	0,078	10,701	1136,76
12	0	0,078	19,827	557,31
13	0	0,093	6,052	1551,97
14	0	0,062	30,264	688,304
15	0	0,062	9,952	648,773
16	0	0,062	10,67	942,475
17	0	0,093	16,146	1340,56
18	0	0,078	5,413	1565,73
19	0	0,093	10,561	1193,92
Min time (s)	**0,000**	**0,046**	**5,413**	**557,310**
Max time (s)	**0,000**	**0,156**	**30,264**	**2663,280**
Mean time (s)	**0,000**	**0,078**	**11,938**	**1135,078**
Min time (mn)	**0,000**	**0,001**	**0,090**	**9,289**
Max time (mn)	**0,000**	**0,003**	**0,504**	**44,388**
Mean time (mn)	**0,000**	**0,001**	**0,199**	**18,918**

TABLE 4.5 – Résultats pour l'algorithme à base de triplets, avec élimination de solutions dominées

Data ID	5 jobs	10 jobs	15 jobs	20 jobs	25 jobs	30 jobs	35 jobs	40 jobs	45 jobs	50 jobs
0	0	0	0	0	0	0	0	0	0,015	0
1	0	0,015	0	0	0	0	0,015	0	0	0,015
2	0	0	0	0	0	0	0	0,015	0,015	0
3	0	0,015	0,078	0	0	0	0	0	0	0,015
4	0	0	0	0	0	0,015	0	0	0,015	0
5	0	0	0	0	0	0	0	0,015	0	0
6	0	0,015	0	0,015	0	0	0	0	0	0,015
7	0	0	0,015	0	0	0,015	0	0	0	0
8	0	0	0	0	0	0	0,015	0	0	0
9	0	0,015	0	0	0	0	0	0	0	0,015
10	0	0	0	0	0,015	0	0	0	0,015	0
11	0	0,031	0	0	0	0	0	0	0	0,031
12	0	0	0	0,015	0	0	0	0,015	0,015	0
13	0	0,015	0	0	0	0	0	0	0	0,015
14	0	0	0,14	0,046	0	0	0,015	0	0,015	0
15	0	0,015	0	0	0	0	0	0,093	0	0,015
16	0	0	0	0	0	0	0	0,015	0,015	0
17	0	0	0	0	0	0	0	0	0	0
18	0	0,015	0	0	0	0,015	0	0	0	0,015
19	0	0	0	0	0	0	0	0	0,514	0
Min time (s)	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**
Max time (s)	**0,000**	**0,031**	**0,140**	**0,046**	**0,015**	**0,015**	**0,015**	**0,093**	**0,514**	**0,031**
Mean time (s)	**0,000**	**0,007**	**0,012**	**0,004**	**0,001**	**0,002**	**0,002**	**0,008**	**0,031**	**0,007**
Min time (mn)	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**
Max time (mn)	**0,000**	**0,001**	**0,002**	**0,001**	**0,000**	**0,000**	**0,000**	**0,002**	**0,009**	**0,001**
Mean time (mn)	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,000**	**0,001**	**0,000**

TABLE 4.6 – Résultats pour l'algorithme *Branch & Bound* (référence)

Data ID	5 jobs	10 jobs	15 jobs	20 jobs	25 jobs	30 jobs	35 jobs	40 jobs	45 jobs	50 jobs
0	390	594	901	1087	1245	1571	2011	2192	2560	2584
1	428	435	938	1157	1342	1585	2094	2259	2476	2631
2	398	585	781	1106	1495	1572	1848	2373	2125	2471
3	403	657	705	1234	1276	1960	1881	2121	2600	2874
4	364	738	959	1153	1407	1770	2117	2039	2407	2847
5	287	822	960	1128	1359	1657	1881	2072	2439	2897
6	441	552	992	1136	1521	1639	1914	2303	2440	2720
7	366	446	891	1290	1206	1644	2185	2130	2449	3000
8	417	647	868	1263	1354	1842	1928	2275	2396	2712
9	326	561	917	1146	1249	1681	1773	1989	2418	2851
10	395	591	897	1010	1327	1568	1704	2271	2270	2658
11	342	673	908	1256	1237	1708	2207	2206	2315	2866
12	326	626	816	1313	1415	1673	1943	2287	2582	2677
13	357	577	834	955	1377	1686	1967	2108	2676	2419
14	307	573	852	1252	1333	1470	1934	2358	2379	2762
15	269	615	919	1198	1492	1656	1860	2131	2787	2665
16	330	608	835	1139	1405	1704	1995	2213	2673	2928
17	305	512	865	1122	1327	1697	2042	2366	2398	2744
18	434	533	880	1209	1494	1872	2015	2053	2586	2681
19	277	575	876	1189	1379	1639	2017	2124	2704	3009

TABLE 4.7 – Valeurs optimales du critère pour toutes les instances étudiées

	5 jobs	10 jobs	15 jobs	20 jobs	25 jobs	30 jobs	35 jobs	40 jobs	45 jobs	50 jobs
Pareto (Extract)	0,000	0,028	1,922	87,006						
Pareto (Set)	0,000	0,020	1,200	54,088						
Triple	0,001	0,906								
Triple & Pareto	0,000	0,078	11,938	1135,078						
Branch and Bound	0,000	0,007	0,012	0,004	0,001	0,002	0,002	0,008	0,031	0,007

Temps d'exécution moyens

	5 jobs	10 jobs	15 jobs	20 jobs	25 jobs	30 jobs	35 jobs	40 jobs	45 jobs	50 jobs
Pareto (Extract)	0,000	0,015	1,107	60,559						
Pareto (Set)	0,000	0,015	0,826	42,354						
Triple	0,000	0,405								
Triple & Pareto	0,000	0,046	5,413	557,310						
Branch and Bound	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000

Temps d'exécution minimaux

	5 jobs	10 jobs	15 jobs	20 jobs	25 jobs	30 jobs	35 jobs	40 jobs	45 jobs	50 jobs
Pareto (Extract)	0,000	0,046	4,492	149,900						
Pareto (Set)	0,000	0,031	2,823	82,524						
Triple	0,015	1,294								
Triple & Pareto	0,000	0,156	30,264	2663,280						
Branch and Bound	0,000	0,031	0,140	0,046	0,015	0,015	0,015	0,093	0,514	0,031

Temps d'exécution maximaux

TABLE 4.8 – Synthèse des mesures des temps d'exécution pour tous les algorithmes

Conclusion

Après avoir réalisé un état de l'art sur la manière d'appliquer des méthodes exactes à diverses problématiques d'ordonnancement, différents types d'algorithmes exponentiels résolvant le problème $F_3||C_{max}$ ont été étudiés de manière détaillée, de l'analyse de leur pseudo-code à la mesure de leurs performances, en passant par l'estimation de leur complexité et leur implémentation en C++. Bien que les choix retenus en termes de structures de données et de progression algorithmique soient en adéquation avec le fonctionnement théorique des méthodes implémentées, les performances obtenues sont nettement inférieures à celles de l'algorithme référence de type *Branch & Bound*.

D'un point de vue plus personnel, ce Projet de Fin d'Études a été pour moi très enrichissant à plusieurs niveaux. D'une part, réaliser l'analyse d'un grand nombre d'algorithmes exacts très différents les uns des autres, mettant en jeu des mécanismes et des structures de données d'un très haut niveau algorithmique, auquel je n'avais jamais été confronté auparavant, m'a permis de progresser, non seulement pour comprendre des méthodes complexes, mais aussi pour évaluer leur complexité au pire cas. D'autre part, j'ai pu me rendre compte de la difficulté de passer d'un algorithme qui fonctionne sur le papier, à un programme à la fois conforme aux instructions du pseudo-code et performant.

Bibliographie

[1] Christophe Lenté, Mathieu Liedloff, Ameur Soukhal, and Vincent T'kindt. Exponential algorithms for scheduling problems. Technical report, Universités de Tours et d'Orléans, 2014.

[2] Christophe Lenté, Mathieu Liedloff, Ameur Soukhal, and Vincent T'kindt. On an extension of the sort & search method with application to scheduling theory. Technical report, Universités de Tours et d'Orléans, 2012.

[3] Christophe Lenté, Mathieu Liedloff, Ameur Soukhal, and Vincent T'kindt. Complexité au pire des cas d'algorithmes exponentiels pour des problèmes de séquencement. Support de présentation (diaporama).

[4] H.T. Kung, F. Luccio, and F.P. Preparata. On finding the maxima of a set of vectors. *Journal of the Association for Computing Machinery*, 22(4) :469–476, October 1975.

[5] Talel Ladhari and Mohamed Haouari. A computational study of the permutation flow shop problem based on a tight lower bound. *Computers & Operations Research*, 32(7) :1831–1847, July 2005.

[6] Samia Kouki. *Algorithmes Parallèles et Distribués pour la résolution du Problème de Flow Shop de Permutation*. PhD thesis, Université de Sfax, 2015. Code C fourni avec le document.

[7] Christophe Lenté, Mathieu Liedloff, Vincent T'kindt, and Lei Shang. Flowshop over 3 machines. Technical report, Universités de Tours et d'Orléans, 2015.

[8] http://www.cplusplus.com/reference/. Documentation des entêtes standard du C++ (norme).

[9] https://msdn.microsoft.com/en-us/library/cscc687y%28v=vs.110%29.aspx. Documentation des entêtes standard du C++ (implémentation Visual Studio 2012).

[10] Stephen Clamage. Mixing c and c++ code in the same program. http://www.oracle.com/technetwork/articles/servers-storage-dev/mixingcandcpluspluscode-305840.html, feb 2011.

Algorithmes exponentiels en ordonnancement

Département Informatique
5e année
2014-2015

Rapport de Projet de Fin d'Études

Résumé : Ce projet de fin d'études est consacré à l'étude d'algorithmes exponentiels dans le cadre des problématiques d'ordonnancement. Tout d'abord, un état de l'art est proposé, permettant de mieux appréhender certaines grandes familles d'algorithmes exponentiels, en montrant comment de telles techniques peuvent être appliquées à divers problèmes d'ordonnancement. Ensuite, une étude approfondie de plusieurs algorithmes exacts résolvant un problème de flowshop à trois machines est menée, de l'analyse du pseudo-code aux résultats expérimentaux, en passant par la justification des choix d'implémentation.

Mots clefs : projet de fin d'études, algorithmes, exponentiels, ordonnancement

Abstract: This end-of-studies project is dedicated to the study of exponential algorithms in the context of scheduling problematics. First, a state of the art is proposed, for a better understanding of some large families of exponential algorithms, showing how such techniques can be applied to various scheduling problems. Next, a detailed study of several exact algorithms solving a three-machine flowshop problem is conducted, from the analysis of the pseudo-code to the experimental results, through the justification of implementation choices.

Keywords: end-of-studies project, algorithms, exponential, scheduling

---

Encadrants
Lei Shang
lei.shang@univ-tours.fr
Vincent T'Kindt
vincent.tkindt@univ-tours.fr
Christophe Lenté
christophe.lente@univ-tours.fr

Étudiant
Pierre-Antoine Morin
pierre-antoine.morin@etu.univ-tours.fr

DI5 2014-2015

École Polytechnique
Université François Rabelais
Tours