Ordonnancement de tâches : Modified Critical Path et déploiement sur la Cloud AWS

Vuacheux Hugo
CentraleSupelec
Paris, France
hugo.vuacheux@student-cs.fr

Deodato V. Bastos Neto
CentraleSupelec
Paris, France
deodato.vasconcelos-bastos@student-cs.fr

I. Introduction

L'orchestration des tâches est un problème clé en optimisation, trouvant une résonance dans divers domaines d'application. Elle joue un rôle essentiel dans la production industrielle, où une gestion efficace des tâches permet d'optimiser la productivité et de minimiser les temps d'arrêt par exemple. Dans le domaine des algorithmes informatiques, elle contribue à améliorer les performances des systèmes en assurant une exécution fluide et parallèle des processus. Enfin, en gestion de projets où une planification rigoureuse des tâches est indispensable pour le respect de délais imposés et optimiser l'utilisation des ressources disponibles.

Supervisé par Aneo, une entreprise de conseil spécialisée en HPC, Cloud, Product Design, R&D et IA/ML, nous avons, dans le cadre de notre cursus, développé un algorithme heuristique de notre choix afin de créer un ordonnancement de tâches sous contraintes. Nous avons ensuite déployé notre algorithme sur le cloud AWS, dans un environnement de simulation haute performance, avec lequel nous avons dû nous familiariser.

Comme nous allons le développer par la suite, nous nous intéresserons à l'algorithme "Modified Critical Path", qui tire son principe d'orchestration du chemin critique, tout en y ajoutant des fonctionnalités pour mieux refléter la réalité. La version la plus aboutie intègre une allocation dynamique de ressources hétérogènes et limitées, dont le nombre peut varier au cours du temps, simulant ainsi les contraintes du cloud. Elle prend également en compte le coût de communication interprocesseurs, qui englobe différentes limitations liées à l'architecture matérielle du cloud. L'hétérogénéité des processeurs se matérialise par deux catégories : ceux qui ne peuvent exécuter que des tâches nécessitant une quantité de mémoire inférieure à un seuil arbitrairement défini, et ceux capables d'exécuter l'ensemble des tâches, sans restriction de mémoire.

A. Etat de l'art

Un algorithme d'ordonnancement est une méthode permettant d'attribuer des tâches à des ressources tout en respectant les contraintes de dépendance et de capacité, afin d'optimiser un critère donné comme le temps total d'exécution, l'utilisation des ressources ou le coût. Les familles d'algorithmes d'ordonnancement incluent les heuristiques, les métaheuristiques, les méthodes exactes et l'apprentissage automatique [1, 7].

Les méthodes heuristiques représentent des approches de résolution qui ne permettent pas de garantir l'optimalité du résultat obtenu [1]. Il s'agit de méthodes de résolution rapides, fondées sur des règles de priorisation, visant à produire une solution approximative dans un délai réduit. Les heuristiques courantes incluent HEFT (Heterogeneous Earliest Finish Time), PEFT (Predict Earliest Finish Time), Min-Min & Max-Min scheduling, List Scheduling comme le Critical Path Scheduling et le Suffrage Algorithm [5, 9, 11].

Les métaheuristiques constituent des améliorations des heuristiques, combinant des algorithmes aléatoires et des recherches locales afin d'explorer efficacement l'espace des solutions. Elles permettent d'obtenir des solutions approximatives de haute qualité pour des problèmes d'optimisation complexes, tels que l'ordonnancement des tâches [10]. Elles améliorent une solution existante en modifiant légèrement l'ordonnancement, assurent un équilibre entre exploration et exploitation pour éviter les minima locaux, s'adaptent aux contraintes complexes (dépendances, ressources limitées), introduisent une part d'aléatoire dans l'affectation des tâches et offrent des performances optimales sur des instances NP-difficiles pour lesquelles les méthodes exactes sont inapplicables. Les métaheuristiques courantes comprennent le Recuit Simulé, les Algorithmes Génétiques, la Recherche Tabou et l'Optimisation par Colonies de Fourmis [3, 6, 8].

Les méthodes exactes représentent des algorithmes qui garantissent l'obtention d'une solution optimale en explorant l'ensemble des solutions possibles d'un problème d'optimisation sous contraintes [4]. À la différence des heuristiques et des métaheuristiques, elles ne recourent pas à des approximations et assurent la meilleure solution possible, sous réserve de leur applicabilité dans un délai raisonnable. Elles offrent une solution optimale garantie, conviennent aux problèmes de petite à moyenne taille présentant des contraintes complexes et fournissent une borne inférieure pour la comparaison d'autres approches,

mais sont confrontées à une explosion combinatoire entraînant un temps de calcul exponentiel pour les problèmes de grande taille, une consommation mémoire importante et une dépendance aux solveurs mathématiques. Les méthodes exactes courantes comprennent la Programmation Linéaire en Nombres Entiers, Branch & Bound et Branch & Cut.

D'autres familles d'algorithmes pour l'ordonnancement incluent la Programmation Dynamique, qui décompose le problème en sous-problèmes plus petits et stocke leurs solutions, et l'Apprentissage Automatique, qui utilise un ensemble de données pour construire une solution approximative [1, 2]. Des exemples d'approches par apprentissage automatique comprennent Hybrid Weighted Ant Colony Optimization (HWACO), Enhanced Ordinal Optimization et Fuzzy Self-Defense Algorithm (FSDA). HWACO compare l'efficacité, le makespan et les coûts de l'optimisation par colonies de fourmis pour l'ordonnancement.

B. Formalisation de notre Problème

La formalisation mathématique de notre problème est la suivante :

Nous considérons un graphe acyclique orienté $G = (T, A)$, où chaque nœud T_i représente une tâche et chaque arête A_{ij} représente une dépendance entre les tâches. Chaque tâche T_i est caractérisée par un ensemble de prédécesseurs Dep_j où j appartient à l'ensemble des tâches précédentes ; une durée d'exécution D_i et une contrainte mémoire Mem_i. De plus nous disposons de deux types de ressources et du seuil arbitrairement défini LMem :

- N_1 ressources de type 1, notées $R1_k$ avec $k \in [0, N_1 - 1]$, qui ne peuvent exécuter une tâche T_i que si $\text{Mem}_i < \text{LMem}$.
- N_2 ressources de type 2, notées $R2_k$ avec $k \in [0, N_2 - 1]$, qui peuvent exécuter toutes les tâches sans restriction de mémoire.

L'objectif est de développer un algorithme MCP afin de minimiser le makespan, c'est-à-dire le temps total d'exécution du graphe, tout en prenant en compte les contraintes ci-dessus soit :

$$\min_{(R1_k, R2_p)_t,} \text{Makespan}(G, \text{LMem}, \text{Ccost}) \qquad (1)$$

où Ccost est le coût de communication entre les ressources différentes.

II. L'algorithme MCP

L'idée principale de notre algorithme repose sur le concept du chemin critique, qui représente le plus long chemin de dépendance dans un projet. Le chemin critique est constitué des tâches qui, si retardées, retarderaient l'ensemble du projet. Il détermine donc la durée minimale nécessaire à l'exécution de toutes les tâches. Cette notion est cruciale car l'objectif est de réduire le makespan, c'est-à-dire la durée totale du projet. En effet, le makespan ne peut pas être inférieur à la durée du chemin critique. Par conséquent, un makespan proche de la durée du chemin critique signifie que notre ordonnancement est proche d'une solution optimale. Cette approche nous permet d'obtenir une planification efficace et bien adaptée aux contraintes du problème.

La construction de notre algorithme s'organise en deux blocs distincts. Le premier bloc consiste en la création d'une file de priorité pour les tâches de notre graphe, où chaque tâche est classée en fonction de son importance et de ses dépendances. Le deuxième bloc concerne l'attribution des tâches aux processeurs, en prenant en compte les contraintes imposées, telles que la capacité mémoire des ressources et les dépendances entre les tâches.

A. Bloc 1 : Calcul du "Latest Start" et file de priorité

L'objectif de notre algorithme est de prendre en entrée un fichier JSON contenant des tâches ainsi que leur durée d'exécution, leurs mémoire et leurs dépendances, puis de renvoyer l'ordre d'exécution des tâches qui minimise la makespan.

Étape 1 : Initialisation de la liste

D'abord, définissons la notion de "Latest Start" d'une tâche i, noté LS(i). Il représente le moment le plus tard auquel la tâche i peut être débutée sans retarder l'achèvement du projet global. Pour commencer, chaque tâche du graphe est initialisée avec un Latest Start égal à 0.

Étape 2 : Tri topologique basé sur l'algorithme de Kahn

Ensuite, nous effectuons un tri topologique basé sur l'algorithme de Kahn. Cette étape repose sur un parcours en largeur et vise à garantir que chaque nœud est traité après tous ses prédécesseurs. Le choix de cet algorithme de tri n'est pas unique ; un autre type de tri aurait pu être utilisé. Une fois les tâches du graphe triées, nous avons l'assurance que le parcours de la liste obtenue respecte l'ordre des dépendances. Il reste alors à optimiser ce tri initial en tenant compte du critère du chemin critique, ce qui conduit à l'étape suivante.

Étape 3 : Calcul du Latest Start

L'étape précédente fournit un premier ordre des tâches sans prendre en compte leur durée. L'objectif ici est de déterminer le moment ultime auquel une tâche doit impérativement se débuter afin de ne pas retarder l'exécution des tâches suivantes, tout en respectant les dépendances. La formule utilisée est la suivante :

$$\text{LS}(i) = \min\left(\text{LS}(j) - D(i)\right), \quad j \in \text{Succ}(i)$$

Cette relation garantit que la tâche i prend en compte le dernier instant où l'une de ses tâches suivantes doit commencer, car elle ne peut pas finir après ce moment.

Ainsi calculé, le Latest Start permet d'identifier les tâches critiques. Une tâche est dite critique si sa marge totale est nulle, c'est-à-dire si le moment le plus tardif auquel la tâche i peut être débutée sans retarder l'achèvement du projet est égal au moment le plus tôt où elle peut commencer (Earliest Start) :

$$\mathrm{LS}(i) = \mathrm{ES}(i)$$

où ES(i) est le "Earliest Start". L'ordre d'orchestration final est donc récupéré à l'étape suivante.

Étape 4 : Remplissage de la file de priorité

L'objectif ici est d'extraire les tâches selon leur `LS` croissant tout en respectant l'ordre topologique (dépendance). Cette étape permet de prendre en compte le chemin critique tout en respectant les contraintes de dépendance. On priorise ainsi l'allocation d'une tâche du chemin critique aux processeurs.

B. Bloc 2 : Attribution des tâches aux processeurs et création de l'ordonnancement

L'ordonnancement des tâches doit respecter deux critères principaux. Le premier étant le respect des dépendances : chaque tâche doit être exécutée après la fin de ses prédécesseurs. Le second étant le respect de la disponibilité d'un processeur : une tâche peut être affectée à un processeur uniquement si celui-ci est libre à ce moment et chaque processeur ne peut exécuter qu'une seule tâche à la fois. Ensuite pour le choix de quelle tâche sur quel processeurs, notre algorithme MCP regarde une série de conditions. Il procède jusqu'a ce que le file de priorité soit vide en regardant succesivement les conditions suivantes :

1. Avant l'affectation d'une tâche à un processeur, il vérifie que les tâches précédentes sont terminées et tient compte du temps de fin le plus tardif parmi toutes les tâches dépendantes.

2. Ensuite si possible, la tâche est affectée à un processeur déjà utilisé par une tâche qui lui est dépendante afin de minimiser les coûts de communication.

3. Si aucun processeur utilisé par une tâche dépendante n'est disponible, l'algorithme cherche parmi les processeurs libres.

4. Si aucun processeur libre n'est disponible, on choisit le processeur qui se libère le plus tôt, en fonction des temps de fin des tâches en cours d'exécution.

5. Une fois affectée à un processeur, la tâche est créée avec ses temps de début et de fin, ainsi que l'identification du processeur auquel elle est assignée.

C. Ajouts de fonctionnalités

La description de l'algorithme ci-dessus correspond à celui du classique "Critical Path" si l'on omet la préférence de processeur pour les tâches dépendantes. Nous le nommons "Modified Critical Path" en raison des fonctionnalités supplémentaires que nous lui apportons.

IIC.1 La disponibilité dynamique des cœurs

La disponibilité dynamique des cœurs modélise un phénomène courant dans le Cloud computing en permettant une réattribution dynamique des tâches, offrant ainsi de l'élasticité à notre algorithme. Le nombre de ressources de chaque type (type 1 et 2) peut ainsi varier dans le temps. Notre algorithme peut simuler le retrait de ressources, soit en cas de panne, soit en raison d'une diminution des ressources allouées. Cela est particulièrement intéressant, car il ne faut pas oublier que l'on paye le nombre de ressources utilisées sur le Cloud. On peut ainsi envisager un système où la limitation du budget réduit le nombre de cœurs ou bien si l'on diminue les ressources allouées en fonction des besoins pour éviter les coûts liés à des processeurs inutilisés. Inversement, pour paralléliser le travail, on peut augmenter le nombre de ressources allouées afin de réduire le temps de travail.

II.C.2 Hétérogénéité des ressources

Nous avons défini deux types de ressources : les processeurs capables d'exécuter des tâches nécessitant une grande mémoire (type 2) et ceux adaptés aux tâches moins exigeantes en mémoire. L'attribution de la mémoire à chaque tâche étant aléatoire du point de vue de l'algorithme, une nouvelle contrainte s'ajoute à celle du respect de l'ordre d'orchestration défini dans le premier bloc de notre algorithme : il faut également affecter chaque tâche au processeur adéquat. Cette contrainte peut allonger le makespan si le nombre de processeurs de type 2 est insuffisant par rapport au nombre qu'il faudrait pour une allocation optimal. Ainsi, l'optimisation du makespan peut être limitée par un facteur critique : le nombre de ressources de type 2. En effet, soit leur nombre est insuffisant, soit la limite mémoire LMem est trop faible par rapport au nombre de processeurs de type 2, ce qui empêche le traitement prioritaire des tâches du chemin critique.

II.C.3 Prise en compte des coûts de communication

Un coût de communication est ajouté lorsque deux tâches dépendantes sont exécutées sur des ressources différentes. Ce coût représente une pénalisation temporelle prenant en compte les délais d'accès à la mémoire, la latence et les frais de communication entre processeurs. Il est appliqué uniquement si le gain de temps global obtenu en changeant de processeur est inférieur aux coûts engendrés par cette migration, rendant ainsi la préférence de processeur moins avantageuse que la migration de tâche.

D. Limitations et amélioration possibles

Avec plus de temps, nous aurions pu approfondir la complexité de notre algorithme. Quelques pistes d'amélioration incluent :

II.D.1 Détermination du nombre de ressources au cours du temps

Une des faiblesses de notre algorithme est que, bien que nous bénéficiions d'une certaine élasticité en modulant le nombre de type ressources dans le temps, cela implique de devoir spécifier ces ressources en entrée de l'algorithme. Ainsi, l'adaptation est quelque peu 'faussée', car nous connaissons, d'une certaine manière, les aléas à l'avance. Une amélioration pourrait consister à implémenter ces changements de manière dynamique, en parallèle avec la progression de l'ordonnancement, de sorte que, même si cette allocation est aléatoire, elle ne soit pas prédéfinie comme un paramètre d'entrée dans notre algorithme. Alternativement, une autre amélioration serait de construire une fonction annexe qui analyserait le nombre de ressources nécessaires à un instant t pour rendre l'ordonnancement des tâches optimal en adaptant le nombre de ressources allouées selon un budget ou pour éviter la non utilisation de ressources allouées ; ou bien un combiné des deux. Un cas de figure n'est pas pris en compte ici : si un processeur tombe en panne pendant l'exécution d'une tâche, celle-ci devrait être relancée sur un autre processeur. Dans notre approche, nous avons choisi de laisser la tâche se terminer malgré la panne.

D.2 Meilleur description des coûts de communication

Actuellement, notre coût de communication est une pénalisation unique englobant divers facteurs sans préciser leur nature ni leur contribution respective. Une amélioration possible consisterait à différencier ces coûts en fonction des tâches, en intégrant par exemple des paramètres globaux tels que la latence du réseau et la bande passante, ou des paramètres individuels comme la taille mémoire des tâches prédécesseurs, etc.

III. Déploiement sur le cloud AWS

AWS (Amazon Web Services) est une plateforme de cloud computing proposée par Amazon. Le cloud est un ensemble de serveurs, de stockage, de bases de données, de logiciels et d'autres services informatiques accessibles via Internet. Un fournisseur de cloud, tel qu'AWS, propose de louer ces ressources à un prix proportionnel à la quantité utilisée, plutôt que d'acheter toute l'infrastructure nécessaire. Cette flexibilité permet à une entreprise de se concentrer sur le produit développé, ou à un chercheur sur sa recherche. Dans le cadre de ce projet, nous avons utilisé deux services d'AWS. Le premier est Amazon S3 (Simple Storage Service), un service de stockage d'objets évolutif et durable, permettant de stocker des données de toutes sortes. Ce service fonctionne comme un système de fichiers à distance. Le second est AWS Lambda, un service de calcul sans serveur qui permet d'exécuter du code sans avoir à provisionner ou à gérer des serveurs.

Le service S3 a été utilisé comme source et destination des données. La description du graphe a été lue à partir d'un fichier sauvegardé dans un dossier du cloud, tous les calculs ont été réalisés, puis le résultat a été enregistré dans un dossier du cloud. De plus, pour utiliser le service Lambda, nous devons d'abord créer une couche afin de partager des dépendances (des bibliothèques) entre plusieurs fonctions Lambda. Ensuite, nous créons la fonction avec le code associé et nous lançons le code sur le cloud. Cependant, pour exécuter le code sur le cloud, nous devons adapter le code de manière à ce que Lambda comprenne quel code il doit lancer [1].

Comme mentionné précédemment, les services d'AWS sont très flexibles. Nous pouvons ainsi configurer la taille de la mémoire et disposer d'un CPU compatible. Ainsi, un nœud de calcul peut être très puissant ou très faible, mais même avec des paramètres très faibles, nous pouvons exploiter l'élasticité du cloud et lancer plusieurs nœuds simultanément. Par conséquent, il est possible de trouver un compromis entre le budget, le temps d'exécution et le rendement lors de la modification des paramètres de la fonction Lambda.

La figure I présente un schéma de déploiement. Pour invoquer la fonction, nous avons deux options : la première est l'appel manuel de la fonction via un terminal ou sur le site web, et la deuxième est la configuration d'un déclencheur externe, par exemple, l'exécution de la fonction Lambda lorsqu'un fichier de type *JSON* est créé dans un répertoire de S3.

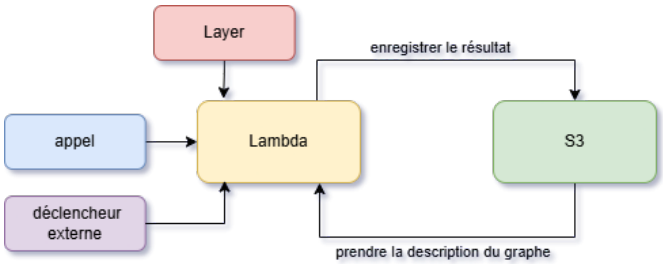


Figure I: Diagrammme du déploiement sur le AWS

En raison des contraintes de permission et de temps, nous n'avons utilisé qu'une seule fonction Lambda. Cependant, nous proposons des stratégies pour gérer les différentes entrées. Premièrement, un téléchargement en parallèle des fichiers permettrait de réduire la latence, spécialement pour les gros volumes de fichiers. Ensuite, plusieurs graphes seraient ordonnés simultanément. Dans ce cas, nous ne parallélisons que les différents graphes, car notre algorithme est plus rapide après le deuxième passage en raison de la sauvegarde du tri basé sur la durée et les dépendances du graphe.

Par ailleurs, nous proposons une autre stratégie : une fonction Lambda principale qui déclenche plusieurs Lambda enfants. La Lambda principale lit l'entrée JSON et déclenche une Lambda enfant en fonction de la taille du graphe. Par exemple, nous pourrions analyser la relation entre la taille du fichier JSON et la quantité de mémoire

[1] https://docs.aws.amazon.com/lambda/latest/dg/python-handler.html

nécessaire pour l'algorithme, puis créer des seuils entre les fonctions. Si une erreur liée à la quantité de mémoire survenait, nous utiliserions une fonction plus puissante. Chaque Lambda enfant gère l'ordonnancement d'une seule taille de graphe et écrit son résultat sur S3.

Enfin, nous pourrions utiliser un autre service d'Amazon, AWS Step Functions, qui permet de créer des flux de travail structurés. De plus, avec AWS Step Functions, il est possible d'utiliser d'autres services d'Amazon, ce qui nous permettrait de réduire les coûts en utilisant des services spécialisés pour chaque partie.

IV. Resultats

Au cours de notre travail, nous avons développé différents algorithmes en ajoutant progressivement des fonctionnalités supplémentaires. Cela nous a permis d'avoir un point de comparaison solide pour observer l'impact de chaque nouvelle fonctionnalité et ajuster notre algorithme.

Au total, nous avons développé cinq versions successives, chacune ajoutant une fonctionnalité par rapport à la version précédente. Ainsi, les trois premières versions ont été développées avec des ressources homogènes. La première version implémentait le classique "Critical Path" avec des préférences définies en fonction des dépendances. La deuxième version a introduit les coûts de communication, puis la troisième a intégré la possibilité de gérer des pannes. Ensuite, nous avons développé une quatrième version, en reprenant la dernière mais cette fois en tenant compte de ressources hétérogènes avec une limite mémoire mais dont le nombre de processeurs était fixe. Enfin, la version la plus élaborée considère une allocation dynamique aléatoire de ressources hétérogènes au fil du temps.

Il convient de noter qu'une fois les algorithmes développés et testés en local, il a été nécessaire d'adapter notre code afin de pouvoir le déployer sur le cloud.

Pour illustrer le fonctionnement de notre algorithme MCP, nous présenterons quelques exemples des différentes versions afin de visualiser l'impact de l'ajout de chaque fonctionnalité.

A. Ordonnancement avec des ressources homogènes

Afin d'obtenir une représentation visuelle de nos ordonnancements et de mesurer l'impact de chaque fonctionnalité, nous présentons un exemple pour les trois premières versions du MCP sur des graphes de petite envergure. Ainsi, pour ces trois premiers algorithmes, nous effectuerons les calculs sur 150 nœuds et 6 processeurs homogènes, avec un coût de communication équivalent à une pénalité temporelle de 1 seconde ajoutée pour le deuxième et le troisième algorithmes. De plus, une panne du processeur 0 sera simulée après 150 secondes pour le troisième algorithme.

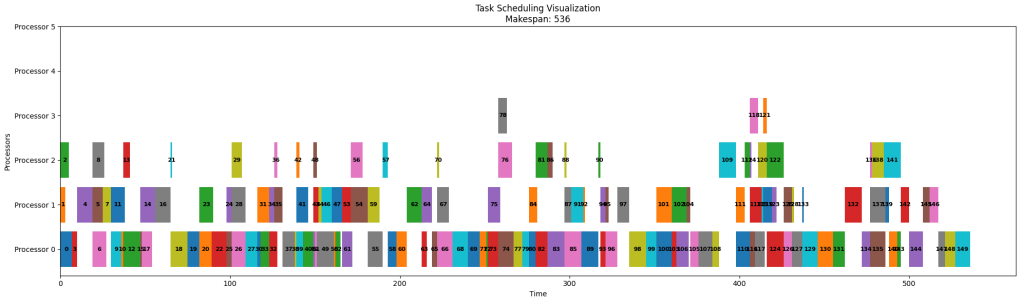


Figure II: Classique "Critical Path"

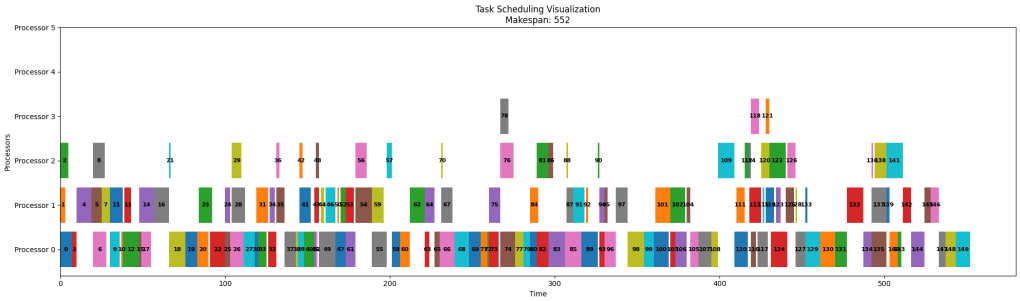


Figure III: Classique "Critical Path" avec prise en compte des coûts de communication

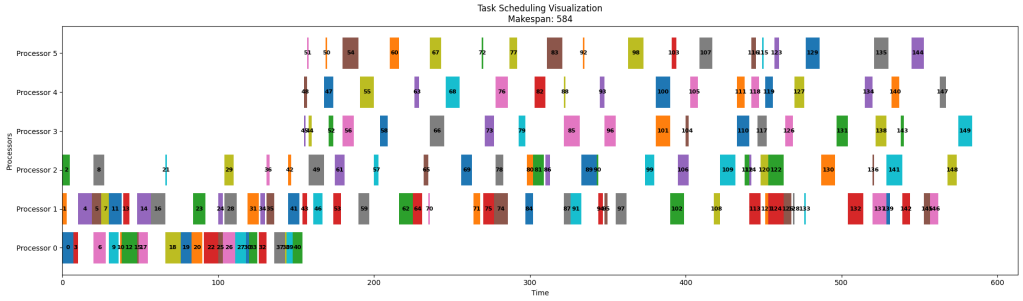


FIGURE IV: CLASSIQUE "CRITICAL PATH" AVEC PRISE EN COMPTE DES COÛTS DE COMMUNICATION ET UNE PANNE

La longueur du chemin critique du graphe commun est de 533. On peut noter l'efficacité de nos algorithmes par trois points principaux. Tout d'abord, le makespan du 'premier algorithme est proche de celui du chemin critique. Ensuite, le fait que nous nous rapprochions de la solution optimale sans avoir utilisé toutes les ressources allouées souligne que nous optimisons l'utilisation des processeurs déjà en travail, permettant ainsi, comme dans le troisième algorithme, un bon rééquilibrage des tâches en cas de panne. Enfin, l'augmentation du makespan dans la deuxième et troisième version est due au coût de communication, qui différé le départ d'une tâche sur un processeur, ce qui est bien observé en comparant les deux premières versions.

B. *Ordonnancement avec des ressources hétérogènes*

Nous illustrons ici un exemple d'ordonnancement pour les algorithmes exploitant des ressources hétérogènes. Nous considérons toujours un graphe composé de 150 nœuds et un ensemble de processeurs répartis en deux types : quatre processeurs de type 1 (indices 0 à 3) et deux processeurs de type 2 (indices 4 et 5).

Le mécanisme de gestion des processeurs varie selon l'algorithme utilisé :

Pour le premier algorithme, nous simulons une panne du processeur 0 après 10 secondes d'exécution. Celui-ci ne se rallume jamais.

Pour le second cas, le planning de fonctionnement des ressources a été défini de manière aléatoire comme suit : les processeurs 1, 2 et 4 fonctionnent en continu, tandis que le processeur 0 subit une panne entre les instants t = 10 et t = 130. De même, le processeur 3 tombe en panne entre t = 150 et t = 330, et le processeur 5 ne démarre qu'après 80 secondes.

De plus, la limite de mémoire entre les processeurs de type 1 et de type 2 a été arbitrairement fixée à 700, tandis que la mémoire allouée aux tâches est attribuée aléatoirement dans l'intervalle [1, 1000]. À titre de comparaison, la longueur du chemin critique du graphe utilisé est de 302.

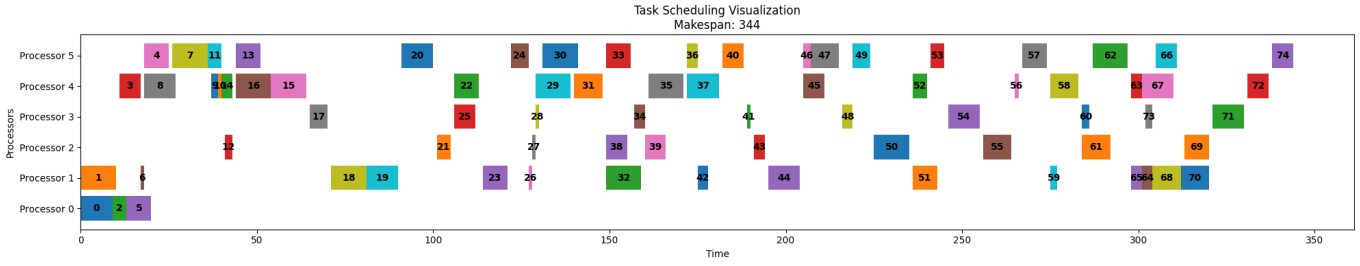


FIGURE V: RESSOURCES HÉTÉROGÈNES, PANNE UNIQUE

La gestion des processeurs étant différente, le premier algorithme, ne pouvant simuler qu'une seule panne, présente moins d'élasticité que le deuxième, dont le nombre de processeurs varie dans le temps selon la méthode décrite précédemment. Il devient alors difficile de comparer les makespans entre eux. Néanmoins, on peut remarquer que leur makespan reste relativement proche de la longueur du chemin critique.

C. *Ordonnancement sur des grands graphes localement*

Nous avons, dans les sections précédentes, illustré le fonctionnement de nos algorithmes sur de petits graphes afin d'en expliciter les principes et de démontrer leur efficacité en obtenant des solutions satisfaisantes. Il reste désormais à évaluer la *scalabilité* de notre approche en testant ses performances sur des graphes de grande envergure, contenant plusieurs dizaines de milliers de nœuds.

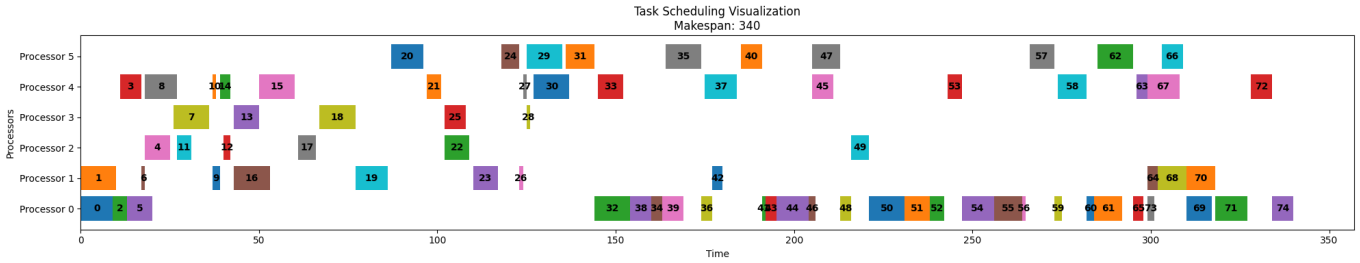


Figure VI: Ressources hétérogènes et allocation dynamique des processeurs

Par exemple si en local, nous considérons un graphe de 10 000 nœuds avec au plus 12 processeurs : sept processeurs de type 1 (indices 0 à 6) et cinq processeurs de type 2 (indices 7 à 11) avec une limite mémoire de 600. La gestion des processeurs au cours du temps est définie comme suit :

- Le processeur 1, 4, 8 et 9 fonctionnent en continu.
- Les autres processeurs subissent des arrêts aux intervalles suivants : le processeur 0 est arrêté entre $t = 17000$ et $t = 24000$, le processeur 2 entre $t = 6000$ et $t = 7000$, le processeur 3 entre $t = 24000$ et $t = 25000$, le processeur 5 entre $t = 40000$ et $t = 50000$, le processeur 6 entre $t = 0$ et $t = 8000$, le processeur 7 entre $t = 28000$ et $t = 35000$, et enfin, le processeur 10 est arrêté à partir de $t = 12000$ jusqu'à la fin de la simulation.

L'ordonnancement des tâches pour le graphe généré est le suivant :

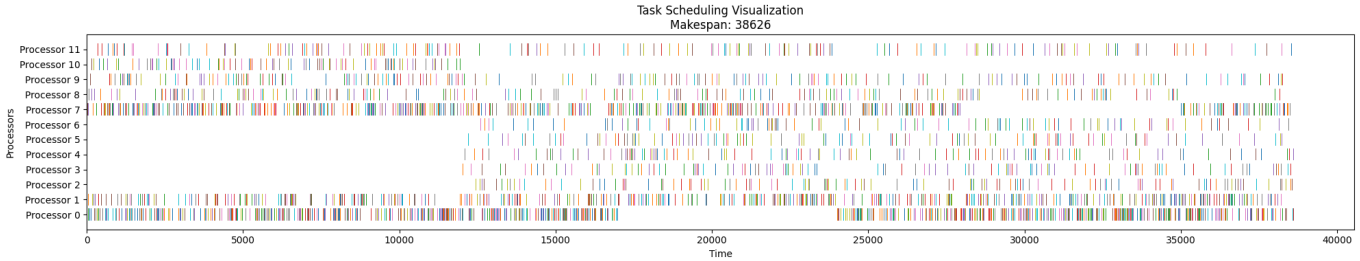


Figure VII: Ressources hétérogènes et allocation dynamique des processeurs sur un graphe de 10000 noeuds

On constate que le rééquilibrage des tâches en cas de panne ou lors de l'ajout d'un nouveau processeur s'effectue correctement. De plus, on observe que seules les ressources strictement nécessaires parmi celles allouées sont utilisées de manière optimale. Cependant, la longueur du chemin critique du graphe utilisé est de 34081 et le temps de calcul est d'environ 32 minutes. Le déploiement sur le cloud est donc nécessaire, car exécuter nos algorithmes localement pour des graphes de plusieurs dizaines de milliers de nœuds serait beaucoup trop coûteux en temps.

D. Ordonnancement sur des grands graphes sur Cloud

Il est évident que faire tourner des algorithmes d'ordonnancement localement pour des graphes de plusieurs dizaines de milliers de nœuds est très complexe, les temps de calcul étant beaucoup trop longs. C'est pourquoi le déploiement de nos codes, adaptés à l'environnement cloud, constitue une solution évidente pour pallier ce problème. La différence est d'ailleurs flagrante.

Nous effectuons un test sur un graphe de 100 000 nœuds généré par le script fourni par l'équipe encadrante (voir l'impact par rapport à notre propre génération dans la partie suivante). Pour ce graphe, nous allons tester pour trois limites mémoire différentes entre les processeurs de type 1 et de type 2 trois distributions aléatoires dynamiques des ressources hétérogènes au cours du temps.

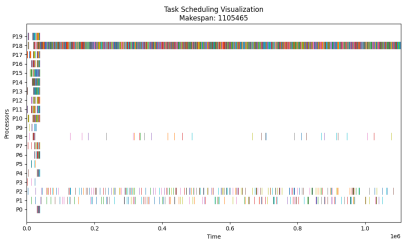
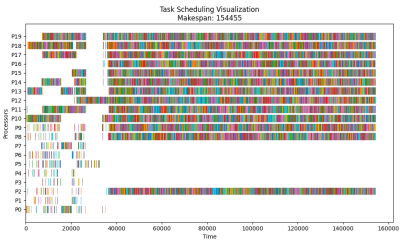
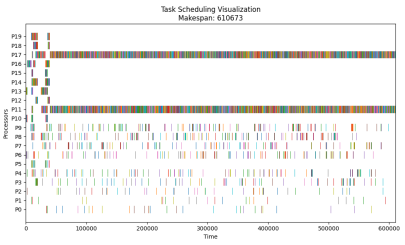
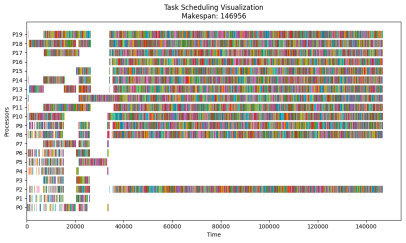
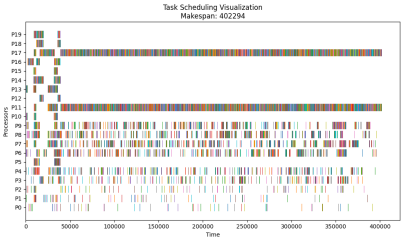
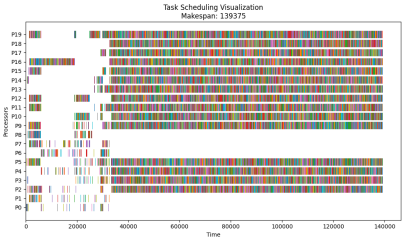
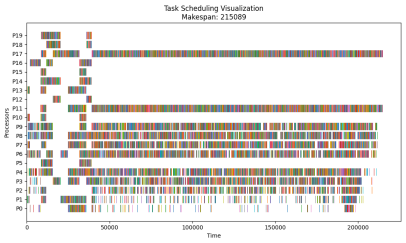
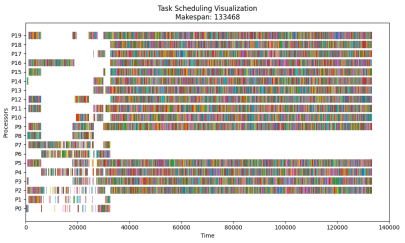
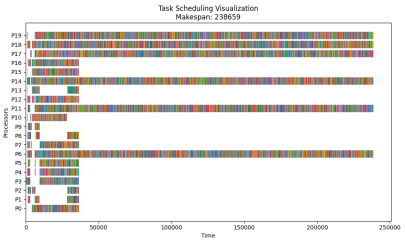
La distribution aléatoire des processeurs au cours du temps suit certaines règles : nous pouvons utiliser au maximum 20 ressources, dont 10 de type 1 et 10 de type 2. Un rééquilibrage aléatoire du nombre de processeurs de chaque type est effectué à 10 instants, eux-mêmes définis aléatoirement, en s'assurant qu'au moins un processeur de chaque type soit en activité.

IV.D.1 Paramétrage de notre test

Nous avons mentionné que nous allions effectuer trois types de répartition dynamique aléatoire, selon trois limites de mémoire, pour notre exemple d'un graphe à 100 000 nœuds. Nous les notons `LMem1`, `LMem2` et `LMem3` les limites mémoires respectivement égales à 256, 512 et 1024. Nous notons respectivement `Mapping(i,j)` et $t_{i,j}$ la répartition des processeurs au cours du temps et le temps d'exécution

d'un algorithme sur le cloud, où i représente l'indice de la limite mémoire et j le j-ème mapping associé à cette limite.

On choisit d'effectuer des réattributions dynamiques des différentes ressources entre les instants $t = 0$ et $t = 40000$. Au-delà de $t = 40000$, le dernier état reste fixe. L'allocation aléatoire des processeurs au cours du temps est reporté en annexe pour ne pas surcharger ici. Nous reportons les résultats dans le tableau ci-dessous :

Mapping(1,1), $t_{1,1} = 28.37$, $Mkp = 1105465$ 	Mapping(1,2), $t_{1,2} = 29.20$, $Mkp = 154455$ 	Mapping(1,3), $t_{1,3} = 28.50$, $Mkp = 610673$ 
Mapping(2,1), $t_{2,1} = 28.07$, $Mkp = 146956$ 	Mapping(2,2), $t_{2,2} = 27.35$, $Mkp = 402294$ 	Mapping(2,3), $t_{2,3} = 27.09$, $Mkp = 139375$ 
Mapping(3,1), $t_{3,1} = 27.06$, $Mkp = 215089$ 	Mapping(3,2), $t_{3,2} = 27.05$, $Mkp = 133468$ 	Mapping(3,3), $t_{3,3} = 26.64$, $Mkp = 238659$ 

Il est certes difficile de tirer des conclusions précises du test précédent, car neuf simulations ne constituent pas un échantillon statistiquement représentatif des différentes configurations possibles des graphes générés avec 100 000 nœuds. Cependant, on retrouve certaines caractéristiques déjà observées avant ce passage à l'échelle.

De plus, l'allocation des processeurs au cours du temps étant totalement aléatoire (tout en respectant certaines règles), le fait de pouvoir manipuler un ensemble de 20 processeurs augmente encore le nombre de tests nécessaires pour obtenir des résultats fiables.

Par manque de temps, nous avons lancé les tests sur le cloud pour 100 000 nœuds en imposant une redistribution dynamique uniquement durant les 40 000 premières secondes, laissant ensuite l'ordonnancement statique pour le reste du temps. Ainsi, le *makespan* reflète imparfaitement la performance réelle de notre algorithme, car, durant les quatre cinquièmes du temps restant, l'ordonnancement des tâches dépend uniquement du dernier état enregistré, qui est aléatoire (par exemple, la case (1,1) atteignant un *makespan* d'un million).

Néanmoins, si l'on se concentre sur ces 40 000 premières secondes, on constate que, comme en local sur des graphes plus petits, notre algorithme s'adapte efficacement en redistribuant les tâches sur les processeurs disponibles et en n'activant ceux-ci que lorsque cela est nécessaire.

De plus, un point extrêmement important est le passage à l'échelle, qui est une réussite. En effet, sur ces neuf simulations, nous obtenons un temps moyen d'exécution de 27,70 secondes. Nous pouvons donc affirmer que nous

avons su adapter notre code et le déployer sur le cloud avec succès, ce qui constituait par ailleurs l'un des enjeux majeurs de ce travail.

Il est clair aussi que l'initialisation du nombre de processeurs conditionne la répartition des tâches en fonction du nombre de nœuds. En effet, à plusieurs reprises, nous avons observé que notre algorithme n'exploite pas toujours l'ensemble des cœurs alloués lorsque le nombre de noeuds n'est pas disproportionné par rapport aux nombre de ressources. Cela constitue une caractéristique très intéressante : l'utilisation de ressources supplémentaires n'entraîne pas nécessairement une réduction du makespan. Ainsi, au lieu d'utiliser toutes les ressources allouées, l'algorithme choisit d'utiliser le nombre minimal de processeurs permettant de minimiser le makespan.

E. Impact de la façon de générer le graphe

Notre algorithme tire sur le principe d'ordonnancement basé sur le chemin critique. Ainsi, pour évaluer la pertinence de notre ordonnancement, il est naturel de comparer le makespan à la taille du chemin critique. Cependant, la manière dont les graphes ont été générés diffère entre ceux que nous avons créés pour tester notre algorithme et ceux fournis par l'équipe encadrante. Cette différence rend la comparaison moins pertinente pour les graphes fournis par l'encadrement que pour les nôtres.

Nous avons généré des graphes acycliques en contrôlant le nombre maximal de dépendances par nœud. De cette façon, en limitant le nombre de dépendances, l'augmentation du nombre de nœuds entraîne naturellement une augmentation de la taille du chemin critique. Il existe donc une forte corrélation entre le nombre de nœuds et la taille du chemin critique dans nos graphes.

En revanche, les graphes fournis par l'équipe encadrante sont générés de manière plus aléatoire, y compris dans la gestion des dépendances. La connexion entre les tâches n'est pas restreinte : les arcs sont choisis aléatoirement, ce qui réduit la corrélation entre le nombre de nœuds et la taille du chemin critique. En effet, pour un nombre donné de nœuds, le nombre de dépendances (c'est-à-dire les arcs du graphe) reliant la première tâche (sans prédécesseur) à la dernière tâche (sans successeur) peut rester relativement faible par rapport au nombre total de nœuds. Ainsi, le ratio entre le nombre de nœuds et la taille du chemin critique est plus élevé dans un graphe où les dépendances sont générées aléatoirement.

Par conséquent, dans notre cas, le makespan de notre ordonnancement dépend bien plus fortement du nombre de processeurs attribués. En effet, l'intérêt de privilégier une tâche appartenant au chemin critique est moins fréquent lorsque la structure du graphe est plus aléatoire, réduisant ainsi l'impact de notre stratégie d'ordonnancement.

V. Conclusion

Pour conclure, nous avons réussi à déployer avec succès notre algorithme *Modified Critical Path* sur le cloud AWS, en l'appliquant à des graphes volumineux de 100 000 nœuds avec un temps moyen d'exécution relativement rapide d'environ 27 secondes. Cet algorithme nous a permis de générer un ordonnancement des tâches prenant en compte plusieurs contraintes : dépendances, gestion mémoire, répartition sur des ressources hétérogènes dont la mise à disposition évoluait de manière aléatoire au cours du temps.

La possibilité d'ajuster dynamiquement le nombre de ressources allouées en intégrant les coûts de communication entre processeurs et la mémoire associée à chaque tâche confère à la version finale de notre algorithme une grande flexibilité.

Ainsi, nous avons pu répondre à la problématique posée : comment ordonnancer un graphe comportant de multiples dépendances dans un environnement contraint, avec une attribution dynamique de ressources hétérogènes, en le déployant sur le cloud tout en minimisant les coûts.

Remerciements

Nous tenons à remercier chaleureusement les membres de l'équipe ANEO pour leur supervision et leur accompagnement tout au long de ce projet. Leur soutien a été précieux, et nous souhaitons particulièrement souligner la qualité des retours réguliers lors des différentes présentations, qui ont permis de guider l'avancement de notre travail.

Nous remercions également l'équipe pour la possibilité de travailler sur le Cloud, une expérience très enrichissante. De plus, la clarté des explications fournies et la réactivité dont ils ont fait preuve chaque fois que nous avons sollicité leur aide ont grandement contribué à la réussite de ce projet.

References

[1] Shahbaz Afzal and Ganesh Kavitha. Load balancing in cloud computing–a hierarchical taxonomical classification. *Journal of cloud computing*, 8(1):1–24, 2019.

[2] Faisal S Alsubaei, Ahmed Y Hamed, Moatamad R Hassan, M Mohery, and M Kh Elnahary. Machine learning approach to optimal task scheduling in cloud communication. *Alexandria Engineering Journal*, 89: 1–30, 2024.

[3] AR Arunarani, Dhanabalachandran Manjula, and Vijayan Sugumaran. Task scheduling techniques in cloud computing: A literature survey. *Future Generation Computer Systems*, 91:407–415, 2019.

[4] Hesham El-Rewini, Hesham H Ali, and Ted Lewis. Task scheduling in multiprocessing systems. *Computer*, 28(12):27–37, 1995.

[5] Kobra Etminani and M. Naghibzadeh. A min-min max-min selective algorihtm for grid task scheduling. In *2007 3rd IEEE/IFIP International Confer-*

ence in Central Asia on Internet, pages 1–7, 2007. doi: 10.1109/CANET.2007.4401694.

[6] Ibrahim Mahmood Ibrahim, AH Radie, Karwan Jacksi, Subhi RM Zeebaree, Hanan M Shukur, Zryan Najat Rashid, Mohammed AM Sadeeq, and Hajar Maseeh Yasin. Task scheduling algorithms in cloud computing: A review. *Turkish Journal of Computer and Mathematics Education*, 12(4):1041–1053, 2021.

[7] Klaithem Al Nuaimi, Nader Mohamed, Mariam Al Nuaimi, and Jameela Al-Jaroodi. A survey of load balancing in cloud computing: Challenges and algorithms. In *2012 Second Symposium on Network Cloud Computing and Applications*, pages 137–142, 2012. doi: 10.1109/NCCA.2012.29.

[8] Fatma A Omara and Mona M Arafa. Genetic algorithms for task scheduling problem. *Journal of Parallel and Distributed computing*, 70(1):13–22, 2010.

[9] KwangSik Shin, MyongJin Cha, MunSuck Jang, JinHa Jung, WanOh Yoon, and SangBang Choi. Task scheduling algorithm using minimized duplications in homogeneous systems. *Journal of Parallel and Distributed Computing*, 68(8):1146–1156, 2008.

[10] Poonam Singh, Maitreyee Dutta, and Naveen Aggarwal. A review of task scheduling based on metaheuristics approach in cloud computing. *Knowledge and Information Systems*, 52(1):1–51, 2017.

[11] Min-You Wu and Daniel D Gajski. Hypertool: A programming aid for message-passing systems. *IEEE transactions on parallel and distributed systems*, 1(3): 330–343, 1990.

VI. Annexe

```
{Mapping 1.1:}
{
  0: ({3}, {17, 11, 13}),
  2728: ({1, 2, 4, 5, 6, 7}, {18, 10, 19}),
  6465: ({0, 1, 2, 4, 5, 6, 7, 8, 9}, {17}),
  9438: ({0, 1, 3, 4, 5, 7, 9}, {16, 11, 12, 15}),
  15619: ({0, 1, 2, 3, 4, 5, 7, 8, 9}, {10, 19, 14}),
  17695: ({8}, {10, 11, 12, 13, 14, 15, 16, 17, 18, 19}),
  22455: ({0, 2, 3, 4, 5, 6, 7, 8, 9}, {10, 11, 12, 13, 14, 15, 16, 17, 18, 19}),
  25471: ({0, 1, 2, 3, 5, 6, 7, 9}, {10, 13, 14, 15}),
  28348: ({0, 4, 6, 7}, {10, 11, 12, 13, 14, 15, 16, 17, 18, 19}),
  39163: ({8, 1, 2}, {18})
}

{Mapping 1.2:}
{
  0: ({0, 4, 6, 8, 9}, {10, 11, 12, 13, 14, 16, 17, 18, 19}),
  193: ({0, 1, 2, 4, 5, 6, 8, 9}, {10, 18, 13}),
  6745: ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, {10, 11, 14, 17, 18, 19}),
  15252: ({0, 1, 2, 5, 6, 7, 8, 9}, {10, 11, 12, 13, 14, 15, 16, 17, 18, 19}),
  15266: ({0, 7}, {11, 13, 17, 18, 19}),
  20397: ({1, 2, 3, 4, 6}, {11, 17, 19, 15}),
  21422: ({0, 1, 2, 3, 5, 6, 7, 8, 9}, {11, 12, 13, 14, 15, 18, 19}),
  26047: ({5}, {12}),
  33210: ({0, 4, 6, 7, 8, 9}, {10, 12}),
  34058: ({8, 9, 2}, {10, 11, 12, 13, 14, 15, 16, 17, 18, 19})
}

{Mapping 1.3:}
{
  0: ({1, 2, 4, 6, 7, 8, 9}, {10, 13}),
  1886: ({0, 2, 3, 4, 6, 8, 9}, {16}),
  8190: ({1, 2, 4, 5, 7, 8, 9}, {10, 11, 13, 14, 15, 17, 19}),
  11634: ({4, 5, 6, 8, 9}, {16, 18, 19, 14}),
  15747: ({3}, {12, 14, 17, 18, 19}),
  20315: ({1, 6}, {17}),
  24796: ({0, 1, 3, 4, 7, 8}, {17, 11, 13}),
  31656: ({0, 1, 4, 5, 6, 7, 8, 9}, {10, 11, 13, 14, 15, 16}),
  35915: ({3, 4, 5, 6, 8}, {12, 14, 16, 18, 19}),
  39144: ({0, 1, 2, 3, 4, 6, 7, 8, 9}, {17, 11})
}

{Mapping 2.1:}
{
  0: ({0, 2, 3, 5, 8}, {14}),
  1033: ({1, 2, 5, 7, 8, 9}, {11, 12, 15, 16, 19}),
  5923: ({0, 1, 2, 4, 6, 7}, {16}),
  17930: ({0, 1, 3, 4, 5, 6, 7, 8, 9}, {19}),
  18806: ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, {10, 12}),
  23778: ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, {19}),
  26180: ({0, 2, 3, 4, 5, 6}, {10, 11, 13, 14, 17}),
  29917: ({0, 2, 3, 5, 6, 7, 9}, {17, 19, 15}),
  30688: ({0, 1, 3, 4, 5, 6, 7, 9}, {19, 11, 13}),
  32563: ({2, 3, 4, 5, 9}, {10, 11, 12, 13, 14, 15, 16, 17, 18, 19})
}

{Mapping 2.2:}
{
  0: ({3, 4, 6, 7, 8, 9}, {10, 11, 12, 13, 14, 15, 16, 17, 18, 19}),
  1230: ({1, 2, 3, 4, 5, 6, 7, 8, 9}, {11, 12, 14, 16, 18}),
  2848: ({8, 9, 2, 6}, {16, 17, 10}),
  3630: ({9, 6, 7}, {17, 14}),
  4186: ({0, 2}, {10, 12, 13, 14, 15, 16, 18}),
  6036: ({0, 1, 2, 4, 5, 6, 9}, {10, 11, 13, 14, 15, 16, 17, 18, 19}),
  6633: ({0, 1, 6, 8, 9}, {10, 11, 12, 13, 14, 15, 16, 17, 18, 19}),
  9445: ({0, 3, 4, 5, 6, 7}, {10, 11, 12, 14, 15, 16, 17, 18, 19}),
  28207: ({0, 1, 2, 3, 4, 5, 6, 7, 8}, {11, 12, 13, 14, 15, 16, 17, 18, 19}),
  36564: ({6}, {11, 14, 17, 18, 19})
}


{Mapping 2.3:}
{
  0: ({0, 2, 4}, {10, 11, 12, 14, 17}),
  5032: ({0, 2, 3, 6, 9}, {10, 11, 12, 13, 14, 15, 16, 18, 19}),
  10256: ({0, 1, 2, 3, 4, 5, 8, 9}, {18}),
  18174: ({1, 6, 7}, {10, 11, 12, 14, 15, 16, 17, 19}),
  23668: ({0, 2, 4, 6, 7, 8, 9}, {11, 12, 14, 15, 16, 17, 18}),
  24966: ({2, 5}, {10, 11, 12, 13, 14, 16}),
  25665: ({0, 1, 3, 4, 5, 6, 7, 8, 9}, {10, 12, 13, 14, 15, 17, 18, 19}),
  27839: ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, {18}),
  33233: ({0, 1, 2, 4, 5, 6, 8, 9}, {10, 14, 15, 16, 17, 18, 19}),
  35908: ({1, 3, 5, 6}, {10, 11, 12, 13, 14, 15, 16, 17, 18})
}

{Mapping 3.1:}
{
  0: ({9, 4, 6, 7}, {10, 11, 13, 14, 15, 16, 17, 18, 19}),
  3505: ({0, 2, 3, 4, 5, 6, 7, 8, 9}, {10, 12, 13, 14, 16, 17, 18}),
  8535: ({3, 4, 5, 6, 9}, {10, 11, 12, 13, 14, 15, 16, 17, 18, 19}),
  10314: ({8, 9, 4}, {17}),
  11193: ({0}, {10, 11, 12, 13, 14, 15, 17, 18, 19}),
  12375: ({0, 1, 2, 3, 4, 5, 8, 9}, {10, 11, 12, 15, 17, 18}),
  20571: ({0, 1, 2, 4, 5, 7, 8}, {10, 12, 13, 14, 15, 16, 17, 18, 19}),
  21341: ({1, 2, 3, 4, 6, 7}, {17, 13, 14, 15}),
  25415: ({0, 1, 2, 4, 5, 6, 7, 8, 9}, {10, 11, 12, 13, 14, 15, 16, 17, 18, 19}),
  26733: ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, {17, 10, 11, 15})
}

{Mapping 3.2:}
{
  0: ({0, 3, 4, 6}, {16, 19, 14}),
  10138: ({3, 4, 5, 8, 9}, {10, 18}),
  12834: ({1, 2, 4, 5}, {16}),
  13793: ({0, 2, 3, 5, 6, 7}, {10, 12, 13, 14, 15, 16, 17, 19}),
  16122: ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, {11, 12, 13, 14, 17, 18}),
  22283: ({1, 2, 3, 5, 7, 8, 9}, {10, 11, 12, 13, 14, 15, 16, 17, 18, 19}),
  25947: ({3, 6}, {16, 19, 15}),
  26080: ({8, 9, 4, 6}, {15}),
  33463: ({8}, {10, 11, 12, 13, 15, 16, 18, 19}),
  37373: ({5, 7}, {18})
}

{Mapping 3.3:}
{
  0: ({2, 3, 4, 6, 7, 8, 9}, {17}),
  2349: ({5}, {10, 12, 15, 16, 17, 18}),
  7117: ({2, 5}, {10, 12, 14}),
  10092: ({4, 5, 6}, {16, 18, 14}),
  11480: ({0, 1, 2, 6, 8}, {13, 14, 16, 17, 18, 19}),
  18921: ({0, 3, 4, 5, 6, 7, 8, 9}, {16}),
  20059: ({0, 2, 3, 4, 5, 6, 7, 8}, {10, 11, 13, 16, 19}),
  25815: ({1}, {16, 18, 10, 12}),
  26096: ({1, 6}, {10, 11, 12, 13, 14, 15, 16, 17, 18, 19}),
  32489: ({0, 1, 2, 3, 4, 5, 6, 7, 8}, {19})
}
```