1 Contexte

Mes travaux de recherche s'inscrivent dans deux différents domaines qui sont les problèmes d'ordonnancements et les automates temporisés.

1.1 Problèmes d'Ordonnancements

Résoudre un problème d'ordonnancement dans un système consiste à déterminer le plan d'exécution de ses composants (date de début d'exécution, sur quelle machine exécuter, etc.) et cela en respectant des contraintes inhérentes ou extérieures au système et en optimisant un certain critère, tel que le temps total d'exécution, retards, nombre de machines utilisées. Les problèmes d'ordonnancement sont en général des problèmes NP-complet.

1.2 Automates Temporisés

Un automate temporisé est un automate muni d'horloges, une horloge est une variable réelle qui croit uniformément dans le temps, ces horloges peuvent être remises à zéro et les transitions de l'automate sont conditionnées par un ensemble de contraintes sur les horloges de la forme $X\#c$ ou $X - Y\#c$ où X, Y sont deux horloges, c un nombre entier et $\# \in \{>, \geq, <, \leq, =\}$.

2 Résumé de thèse

Durant ma thèse , j'ai développé une nouvelle méthodologie pour modéliser et résoudre les problèmes d' ordonnancement. L'essence de cette approche est de modéliser les systèmes à étudier en automates temporisés, tel que tout ordonnancement réalisable correspond à un chemin dans l'automate et tout ordonnancement optimal correspond au plus court chemin dans cet automate.

2.1 Job-shop déterministe

La construction de mon modèle a été basée sur le problème du job-shop. Ce problème est un problème d'ordonnancement qui consiste en un ensemble de jobs à exécuter sur un ensemble de machines, chaque job est constitué d'étapes, où toute étape est caractérisée par la machine sur laquelle elle doit être exécutée et son temps d'exécution sur cette machine. Le but est de construire un ordonnancement qui minimise le temps d'exécution total, ce problème est connu pour être NP-complet.

Pour *modéliser* ce problème je commence par modéliser chaque job par un automate temporisé et pour obtenir le modèle global, il suffit de composer ces automates. En premier lieu, j'ai réussi à montrer que la résolution de ce problème revient à la résolution du problème du plus cours chemin dans l'automate temporisé obtenu, j'ai ensuite adapté les algorithmes d'atteignabilité dans les automates temporisés à la résolution du problème du plus court chemin, de cette manière j'ai réussi à résoudre le problème du job-shop à l'aide d'automates temporisés.

Pour que mon algorithme soit plus efficace j'ai introduit un résultat sur les ordonnancements optimaux, cette notion a été appelée les *ordonnancements non-paresseux*, de la même façon j'ai défini ce qu'on appelle un *chemin non-paresseux* dans un automate temporisé. J'ai montré que pour trouver une solution optimale, il ne suffit d'explorer que les chemins non-paresseux, ce résultat m'a permis de réduire remarquablement l'espace de recherche, pour chaque état on passe d'une structure de données quadratique en le nombre d'horloges de l'automate à un vecteur de la taille du nombre d'horloges, les opérateurs de calcul de successeurs ont aussi été simplifiés. A partir de ce résultat j'ai construit et implémenté des algorithmes de recherche dans lesquels j'ai introduit plusieurs techniques de réduction du nombre d'état à explorer. Pour résoudre des problèmes de grandes taille, j'ai proposé une heuristique que j'ai testé sur un ensemble de benchmarks, les solutions sont obtenues en un temps rapide (dans les 5 mn) et sont entre 5 à 10 % plus grandes que les meilleures solutions obtenues jusqu'à présent par la communautés de recherche opérationnelle en utilisant toutes les techniques déjà connues.

2.2 Job-shop avec préemption

J'ai étendu mon modèle au problème du job shop où la préemption est autorisée (une étape peut être interrompue durant son exécution pour être reprise un peu plus tard). Le modèle d'automate temporisé n'est plus valable dans ce cas, j'ai donc modélisé le problème à l'aide d'automates stopwatch qui sont une extension des automates temporisés dans lesquels les horloges peuvent être gelées dans certains états. Dans ce cas l'automate obtenu est cyclique, et les méthodes de résolution proposées pour le cas non-preemptif ne sont plus valables. J'ai utilisé une propriété des ordonnancements optimaux dans le cas preemptif, j'ai appelé les ordonnancements qui respectent cette propriété *ordonnancements efficaces*, ce résultat permet d'éliminer toutes les preemptions inutiles et de résoudre les conflits entre étapes (deux étapes qui attendent la même machine) selon un ordre de priorité. Dans mon modèle ce résultat ce traduit par l'élimination de tous les cycles de mon automate. J'ai construit un algorithme de recherche qui génère les priorités à la volé et qui ne construit que les ordonnancements efficaces. J'ai testé mon heuristique sur des benchmarks et comparé a une méthode basée sur des algorithmes de satisfaisabilité de contraintes, les résultats obtenus sont en moyenne dans les 5 % supérieurs à l'optimum.

2.3 Ordonnancement d'un graphe de tâche

Dans ce problème on dispose d'un nombre fixe de machines identiques (toute tâche peut être exécutée sur n'importe quelle machine) sur lesquels on doit exécuter un nombre de tâches liées entre elles par des contraintes de précédence, ces contraintes sont données sous la forme d'un graphe.

La première idée a été de modéliser chaque tâche à l'aide d'un automate temporisé, cette façon de faire permet de résoudre le problème mais peut être coûteuse, vu que le nombre d'horloges dans l'automate global est égal au nombre de tâches à ordonnancer. J'ai proposé une autre approche qui consiste à décomposer le graphe de tâche en un ensemble de *chaînes* tel que les tâches appartenants à une même chaîne ne peuvent jamais être actives en même temps. A partir d'une décomposition j'associe à chaque chaîne un automate temporisé, dans ce cas, nous obtenons des automates qui communiquent entre eux (une transition dans un automate peut dépendre de l'état dans lequel se trouve un autre automate). La composition des automates de chaînes se fait en respectant le nombre de machines disponibles, on obtient un automate acyclique dans lequel le résultat de non-paresse reste valable, les algorithmes proposés précédemment peuvent être appliqués, je les ai testé sur un ensemble de benchmarks et comparé à une méthode de branch and bound, les solutions sont très proches de l'optimum et calculées en quelques secondes.

2.4 Ordonnancement avec incertitudes temporelles

Les problèmes précédents sont des problèmes déterministes dans le sens où toutes les données du système à ordonnancer sont connues à l'avance. Je me suis intéressée à un problème *non-déterministe*, le job-shop avec *incertitudes temporelles*. Dans ce cas, les durées d'exécution des étapes ne sont plus connues à l'avance, pour chaque étape on dispose d'un intervalle de temps $[a, b]$ dans lequel l'étape peut terminer son exécution, a est la durée minimale de l'étape et b sa durée maximale. Une instance du problème est une assignation d'une valeur aux durées d'éxecution, pour toute instance j'obtiens un problème déterministe que je sais déjà résoudre.

Une façon de résoudre ce problème est de construire un *ordonnanceur statique*, pour cela je construis une solution pour l'instance *pire cas* (l'instance pour laquelle toutes les étapes terminent à leur temps d'exécution maximale) et j'applique cette solution quelque soit l'instance de mon problème. Cette manière de résoudre le problème est efficace en temps mais peut donner des solutions beaucoup plus mauvaises que la solution optimale de la vraie instance. Mon but a été de construire un *ordonnanceur adaptatif* qui à chaque instant prend ses décisions selon l'état dans lequel se trouve le système. Cela peut être fait de manière *on-line*, lorsqu'une étape finit son exécution on obtient un nouveau problème que j'appelle *problème résiduel*, pour prendre une décision l'ordonnanceur doit résoudre se nouveau problème en fixant les durée d'exécutions de ses étapes. Cet ordonnanceur n'est pas applicable en pratique.

Mon approche a été de construire un ordonnanceur *adaptatif off-line* pour cela je construis avant l'exécution toutes les configurations possibles du système, et à chaque configuration j'associe une fonction qui me donne le temps minimale pour atteindre l'état final du système. Pour construire cette fonction, j'ai appliqué un algorithme de *synthèse* qui est une méthode de calcul des états atteignables en arrière dans un automate temporisé. J'ai appliqué mon approche sur des exemples pour lesquels j'ai généré un grand nombre d'instances, et j'ai comparé mes résultats à un ordonnanceur statique, et aux solutions optimales des instances, les résultats obtenus sont très proches des solutions optimales des instances.