DU CONTRÔLE OPTIMAL ET SOUS-OPTIMAL EN PRÉSENCE D'ADVERSAIRES [1]

Oded Maler *

* CNRS-VERIMAG, *2, av. de Vignate, 38610, Gières, France.*
Oded.Maler@imag.fr
www-verimag.imag.fr/∼maler/

Résumé Cet article présente une esquisse d'un cadre unifié pour poser et résoudre des problèmes de contrôle optimal en présence de perturbations incontrôlées. Après avoir posé le cadre général, nous regardons de près un cas concret où le contrôleur est un ordonnancer et les perturbations sont liées à des incertitudes sur la durée des tâches.

1. INTRODUCTION

Je voudrais profiter de cette occasion pour présenter un article libre de tout "nouveau résultat original", sans prétendre mettre quelque chose de nouveau sous le soleil. Je décrirai ce que je considère être l'essence de nombreuses activités concernées par la mise au point de systèmes fondés sur des modèles mathématiques, sans, je crois, dire quoi que ce soit qui ne soit pas connu dans l'une des nombreuses disciplines et communautés qui s'occupent de ces problèmes. Cependant, je crois que mettre tout ceci sous la forme ici présente a une valeur "synergique" qui va au-delà d'une collection de résultats, algorithmes et définitions informelles. Cet article a deux parties principales. Dans la première je présente une sorte de "théorie spéciale de tout" où la conception de système est vue comme la synthèse d'une stratégie optimale dans un jeu dynamique. Nous commenccons dans la section 2 avec une discussion générale de la manière d'évaluer un système qui est soumis à des perturbations extérieures. Dans la section 3 nous introduisons le modèle générique que nous utilisons, un jeu dynamique "multi-étapes" entre un contrôleur et son environnement. La restriction du problème aux comportements de longueur bornée est le sujet de la section 4 où elle est réduite (pour les systèmes à temps discret) à une optimisation sous contraintes standarde de dimension finie. La technique de résolution classique connue sous le nomme *programmation dynamique* est décrite dan la Section 5, suivie, dans la section 6, par la méthode alternative de recherche heuristique en avant. Dans la deuxième partie de cet article je m'intéresse à un cas concret de cet cadre, le problème d'ordonnancement avec incertitude bornée qui est modélisée comme un jeu discret sur un temps continu utilisant le modèle d'automate temporisé. La section 7 tente de convaincre le lecteur que l'ordonnancement tombe dans la catégorie des problèmes décrite dans la première partie. Le problème de "job shop" est décrit dans la Section 8 avec son schéma de solution traditionnel basé sur l'optimisation combinatoire non-convexe. Dans la Section 9 nous montrerons comment ce problème peut être résolu en utilisant des algorithmes de plus court chemin sur des automates temporisés. La section 10 étend le problème aux taches de durées incertaines, et la section 11 décrit un algorithme de programmation dynamique qui peut trouver des stratégies adaptative qui sont meilleures que les stratégies du pire cas.

Les sujets discutés dans cet article sont traités par différentes disciplines et sous divers titres tels que vérification de systèmes, synthèse de contrôleur, prise de décision séquentielle, théorie des jeux, processus de Markov contrôlés, planification en l'IA, contrôle optimal et par modèle prédictif, algorithmes du plus court chemin, programmation dynamique, optimisation, jeux différentiels etc., chacune avec sa propre terminologie. Par exemple, ce qui est appelé dans un contexte une stratégie, peut être appelé ailleurs une politique, une loi de feed-back, un contrôleur ou une

[1] This research was supported in part by the European Community projects IST-2001-33520 CC (Control and Computation) and IST-2001-35304 AMETIST (Advanced Methods for Timed Systems).

règle de répartition. Pire encore, le même terme peut avoir des sens différents pour des publics différents. J'ai fait de mon mieux pour choisir chaque fois le terme (ou les termes) que je trouvais être le plus approprié pour la discussion et j'ai essayé de ne pas passer d'un terme à l'autre trop souvent. En tout cas je m'excuse pour la gêne potentielle occasionné à ceux qui sont habitués à leur terminologie spécifique. J'ai aussi essayé de ne pas biaiser la description par ma propre expérience de la théorie des automates.

2. OPTIMISATION STATIQUE

Tout au long de ce papier nous nous intéresserons à des situations qui ressemblent à des jeux à deux joueurs. Un joueur, le *contrôleur*, représente le système que nous voulons concevoir tandis que l'autre joueur, *l'environnement*, représente des perturbations externes en dehors de notre contrôle. Le contrôleur choisit des actions $u \in U$, l'environnement choisit $v \in V$ et laisse ces choix déterminer le résultat du jeu. Le contrôleur veut que le résultat soit aussi bon que possible, d'après des critères pré-définis, tandis que l'environnement, à moins que l'on ne soit paranoïaque, est indifférent aux résultats. Pour illustrer la problématique de l'optimisation d'une chose qui ne dépend pas seulement de nos actions nous commencons avec un simple jeu à un coup du type introduit dans le travail classique de von Neumann et Morgenstern. Soit $U = \{u_1, u_2\}$, $V = \{v_1, v_2\}$ et soit le résultat défini comme une fonction $c : U \times V \rightarrow \mathbb{R}$ qui peut être donnée sous la forme d'une table

c	v_1	v_2
u_1	c_{11}	c_{12}
u_2	c_{21}	c_{22}

Nous voulons choisir parmi u_1 et u_2 celui qui minimise c mais comme différents choix de v peuvent conduire à différentes valeurs nous devons spécifier comment prendre ces valeurs en compte. Il y a trois approches génériques pour évaluer notre décision :

– *Pire cas* : Chaque action du contrôleur est évaluée en fonction du pire dénouement qui peut en résulter :

$$u = argmin\ \max\{c(u, v_1), c(u, v_2)\}.$$

.

– *Cas moyen* : L'environnement est modélisé comme un agent stochastique agissant au hasard d'après une distribution de probabilité $p : V \rightarrow [0, 1]$ et les actions du contrôleur sont évaluées selon le fonction d'ésperance de c :

$$u = argmin\ p(v_1) \cdot c(u, v_1) + p(v_2) \cdot c(u, v_2).$$

– *Cas typique* : L'évaluation est réalisée par rapport à un élément fixe de V, disons v_1, qui représente le comportement le plus "typique" de l'adversaire. Cela correspond à renier l'existence de perturbations incontrôlées et le problème est réduit à une optimisation ordinaire :

$$u = argmin\ c(u, v_1).$$

Avant de passer aux jeux dynamiques, notons ce qui arrive quand c est une fonction continue sur des ensembles U et V continues. L'analyse de cas moyen reste dans le cadre standard de l'optimisation continue, c'est-à-dire l'optimisation d'une fonction à valeurs réelles, mais pas l'analyse min-max du pire cas. Ceci peut expliquer partiellement pourquoi dans des domaines tels que l'automatique continue les perturbations stochastiques sont beaucoup plus populaires.

3. JEUX DYNAMIQUES

Un jeu dynamique est un jeu où les joueurs participent à une interaction continue étendue dans le temps. Dans le contexte de l'informatique, le terme systèmes réactifs, inventé par Harel et Pnueli, est utilisé pour désigner de tels objets. Un jeu est caractérisé par un espace d'états X et une règle dynamique de la forme

$$x' = f(x, u, v)$$

déclarant que à chaque instant la valeur "suivante" de x est une fonction de sa valeur actuelle et des actions des deux joueurs. Dans cette partie de l'article nous nous focalisons sur les jeux "synchrones" à temps discret où une telle dynamique est souvent écrite comme une équation de récurrence de la forme

$$x_i = f(x_{i-1}, u_i, v_i)$$

mais nous gardons à l'esprit l'existence d'autres modèles tels que les jeux différentiels sur temps dense définis via

$$\dot{x} = f(x, u, v)$$

ou tel que les jeux plus "asynchrone" où des actions peuvent avoir lieu sur de points arbitraires sur l'axe du temps (déclenchement par des évènements plutôt que par le temps) et où les actions des deux joueurs n'ont pas à avoir lieu simultanément. De tels jeux asynchrones seront utilisés plus tard pour modéliser des problèmes d'ordonnancement.

Nous supposons que tous les jeux commencent par un état initial x_0 et utilisons la notation $\bar{x}$ pour une séquence d'états (trajectoire) $x[0], x[1], \ldots, x[k]$. De même, nous utiliserons $\bar{u}$ et $\bar{v}$ pour des séquences d'actions de joueurs. Le prédicat (contrainte) $B(\bar{x}, \bar{u}, \bar{v})$ dénote le fait que $\bar{x}$ est le comportement du système quand les deux joueurs appliquent les séquences d'actions $\bar{u}$ et $\bar{v}$, respectivement :

$$B(\bar{x}, \bar{u}, \bar{v})\ \textit{ssi}\ x[0] = x_0$$
$$x[t] = f(x[t-1], u[t], v[t])\ \forall t$$

Il est parfois utile de voir le jeux comme un graphe étiqueté orienté dont les noeuds sont les éléments de X et dont les arrêts sont toutes les pairs (x, x') telles que $f(x, u, v) = x'$ pour un certain u et v. La partie

atteignable de ce graphe de transition est le sous-graphe obtenu par restriction à des éléments de X pour lesquels un chemin partant de x_0 existe. Une notation alternative utile pour $B(\bar{x}, \bar{u}, \bar{v})$ est :

$$x[0] \stackrel{u[1],v[1]}{\longrightarrow} x[1] \cdots \stackrel{u[k],v[k]}{\longrightarrow} x[k].$$

Il y a de nombreuses manières d'assigner des mesures de performance à de tels comportements afin de les comparer et de trouver l'optimal. Par exemple, toute fonction $c : X \to \mathbb{R}$ sur des états peut être élevée à une fonction de coût sur des séquences en laissant

$$c(\bar{x}) = \sum_{t=1}^{k} c(x[t])$$

ou

$$c(\bar{x}) = \max\{c(x[t]) : t \in 1..k\}.$$

Une autre mesure utile est le temps minimal pour atteindre un but $F \subseteq X$:

$$c(\bar{x}) = \min\{t : x[t] \in F)\}.$$

Dans le cas plus général la fonction de coût peut aussi prendre en compte les prix associés aux actions du contrôleur $\bar{u}$. La nature des fonctions de coût dépend du domaine d'application. Dans la vérification discrète $c(x)$ est typiquement une fonction de valeur $\{0, 1\}$ indiquant les mauvais états, et vérifier les propriétés d'invariance (c'est-à-dire si le système évite toujours ces mauvais états) revient à vérifier si $c(\bar{x}) = 0$ pour l'extension-$\max$ de c aux séquences. Dans les domaines continus certaines fonctions quadratiques sur $\bar{x}$ ou d'autres normes sont souvent utilisées pour indiquer la distance de la séquence (trajectoire) à une référence. Parfois le choix de la fonction de coût est moins influencé par son adéquation au problème et plus par l'existence d'une méthode d'optimisation correspondante, surtout lorsque l'optimum doit être calculé analytiquement.

4. PROBLÈMES D'HORIZON BORNÉE

Nous allons maintenant nous restreindre aux situations où nous comparerons seulement des comportements d'une longueur finie fixée. Il y a plusieurs raisons pour se concentrer sur les décisions d'horizon bornée. La première est qu'il existe certains problèmes du type "contrôle à cible" ou "plus court chemin", où tous les comportements raisonnables convergent vers un état but en un nombre borné d'étapes (mais par des chemins de coûts différents). Une autre raison est l'intuition de sens commun selon laquelle le plus loin on regarde dans le futur, le moins nos modèles deviennent fiables, et donc il vaut mieux planifier pour un horizon plus court et revoir le plan durant l'exécution (c'est la base du contrôle "model-predictive"). Pour finir, les problèmes d'horizon bornée dans un temps discret peuvent être réduit à des problèmes d'optimisation de dimension finie.

Nous illustrons la formulation du problème pour les situations sans adversaire avec des dynamiques de la forme $x' = f(x, u)$. Dans ce cas nous cherchons une séquence $\bar{u} = u[1], \ldots, u[k]$ qui est la solution du problème d'optimisation sous contraintes

$$\min_{\bar{u}} c(\bar{x}) \text{ subject to } B(\bar{x}, \bar{u}).$$

Ici nous utilisons une fonction de coût fondée seulement sur $\bar{x}$ alors que le fait que $\bar{x}$ est le résultat de la dynamique f sous contrôle u fait partie des *contraintes*. Pour les dynamiques linéaires, spécifiées par $x' = Ax + Bu$, et une fonction de coût linéaire, le problème est réduit à de la programmation linéaire standard. Dans la vérification discrète où les dynamiques et le coût sont définis logiquement, le problème est réduit à une satisfiabilité booléenne (C'est l'essence de la vérification de modèles bornés (BMC)).

Si nous disposons d'une procédure d'optimisation sous contraintes pour le domaine en question, nous pouvons calculer le $\bar{u}$ désiré. Notez qu'en l'absence de perturbations externes, $\bar{u}$ détermine complètement $\bar{x}$ et aucune rétroaction de x n'est nécessaire. La "stratégie" de contrôle est réduite à un "plan" boucle ouverte : à chaque instant t, appliquer l'élément $u[t]$ de $\bar{u}$. Nous aurions pu l'énoncer comme une fonction de rétroaction (stratégie) s définie sur tout $x[t]$ comme $s(x[t]) = u[t + 1]$ mais ce n'est pas nécessaire.

Réintroduisons l'adversaire et utilisons, sans perte de généralité, le critère du pire cas. Nous devons maintenant trouver $\bar{u}$ solution de

$$\min_{\bar{u}} \max_{\bar{v}} c(\bar{x}) \text{ subject to } B(\bar{x}, \bar{u}, \bar{v}).$$

Considérez le cas où $U = \{u_1, u_2\}$ et $V = \{v_1, v_2\}$ qui est représenté pour l'horizon 2 par l'arbre de jeu de la figure 1. Nous supposons, par simplicité, que le coût de chaque comportement est déterminé par son état terminal. Dans ce cas nous pouvons énumérer les 4 séquences de contrôle possibles et calculer le coût qu'elles induisent comme :

$$\begin{aligned}
u_1u_1 &: \max\{c(x_5), c(x_6), c(x_9), c(x_{10})\} \\
u_1u_2 &: \max\{c(x_7), c(x_8), c(x_{11}), c(x_{12})\} \\
u_2u_1 &: \max\{c(x_{13}), c(x_{14}), c(x_{17}), c(x_{18})\} \\
u_2u_2 &: \max\{c(x_{15}), c(x_{16}), c(x_{19}), c(x_{20})\}
\end{aligned}$$

La séquence qui minimise ces valeurs est l'optimal parmi les commandes boucle ouverte possible. En utilisant la rétroaction, cependant, on peut faire mieux. Alors que le choix de $u[1]$ peut être fait sans connaissance de l'action de l'adversaire, le choix de $u[2]$ est fait *après* l'effet de $v[1]$, c'est-à-dire que la valeur de $x[1]$ est connue. Considérez le cas où $u[1] = u_1$ et nous devons choisir $u[2]$. Si, par exemple,

$$\max\{c(x_5), c(x_6)\} < \max\{c(x_7), c(x_8)\}$$

mais

$$\max\{c(x_9), c(x_{10})\} > \max\{c(x_{11}), c(x_{12})\}$$

alors l'optimum est d'appliquer u_1 lorsque $x[1] = x_1$ et u_2 quand $x[1] = x_2$.

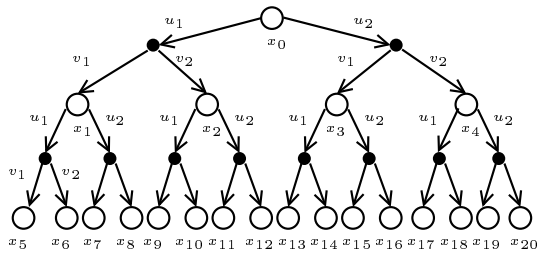

FIG. 1. Un arbre de jeux de profondeur 2.

Une *stratégie de contrôle* est donc une fonction $s : X \rightarrow U$ indiquant au contrôleur quoi faire à chaque étape atteignable du jeu. Le prédicat suivant indique le fait que $\bar{x}$ est le comportement du système en présence de la perturbation $\bar{v}$ quand le contrôleur emploie la stratégie s :

$$\begin{aligned} B(\bar{x}, s, \bar{v}) \text{ ssi } & x[0] = x_0 \\ & u[t] = s(x[t-1]) \ \forall t \\ & x[t] = f(x[t-1], u[t], v[t]) \ \forall t \end{aligned}$$

Trouver la meilleure stratégie (au pire cas) s devient le problème d'optimisation du second-ordre suivant :

$$\min_{s} \max_{\bar{v}} c(\bar{x}) \text{ subject to } B(\bar{x}, s, \bar{v}).$$

Trouver une stratégie optimale est d'habitude beaucoup plus difficile que de trouver une séquence optimale. Les systèmes discrets d'états finis admettent $|U|^{|X|}$ stratégies potentielles et chacune d'entre elles induisent $|V|^k$ comportements de longueurs k. Dans les domaines continus (sur un temps continu), où l'énumération de toutes les stratégies n'est pas une option, une telle stratégie est la solution d'une équation aux dérivées partielles connue comme l'équation de Hamilton-Jacobi-Bellman-Isaacs. Notez que la stratégie n'a pas à être définie sur tout X mais seulement pour des éléments atteignables à partir de x_0 quand le contrôleur emploie cette stratégie.

5. PROGRAMMATION DYNAMIQUE

La programmation dynamique, ou itération à valeur en arrière, est la technique, préconisée par Bellman, pour calculer des stratégies optimales de manière incrémentale. Pour les systèmes discrets l'algorithme est polynômial en la taille du graphe de transitions, ce qui vaut mieux que l'énumération exponentielle de stratégies. Cependant, comme nous le verrons plus tard, ce n'est pas un grand réconfort dans de nombreuses situations où le graphe de transition lui-même est *exponentiel* en le nombre de variables du système.

Nous allons illustrer la programmation dynamique sur le problème de plus court chemin suivant. Un sous-ensemble F de X est désigné comme ensemble cible, et un coût $c(x, u, v)$ est associé à chaque transition. Le coût d'un chemin

$$x[0] \xrightarrow{u[1],v[1]} x[1] \cdots \xrightarrow{u[k],v[k]} x[k]$$

de l'état initial à un état cible est

$$c(\bar{x}, \bar{u}, \bar{v}) = \sum_{t=1}^{k} c(x[t-1], u[t], v[t]),$$

et notre objectif est de trouver la stratégie qui minimise le pire scénario.

La programmation dynamique utilise une fonction auxiliaire (fonction valeur, cost-to-go) $\vec{\mathcal{V}}$: $X \rightarrow \mathbb{R}$ telle que $\vec{\mathcal{V}}\ (x)$ est la performance de la stratégie optimale pour le sous-jeu commenccant à partir de x. Pour les graphes de transition équilibrés cycliques (où tous les chemins qui atteignent un état x à partir de x_0 ont le même nombre de transitions), $\vec{\mathcal{V}}$ admet la définition récursive "en arrière" suivante :

$$\vec{\mathcal{V}}\ (x) = 0 \quad \text{quand } x \in F$$

$$\vec{\mathcal{V}}\ (x) = \min_{u} \max_{v} (c(x, u, v) + \vec{\mathcal{V}}\ (f(x, u, v))).$$

Dans un contexte plus général, comprenant éventuellement des graphes cycliques, la fonction de valeur est le point fixe de l'itération suivante :

$$\vec{\mathcal{V}}_0\ (x) = \begin{cases} 0 & \text{quand } x \in F \\ \infty & \text{quand } x \notin F \end{cases}$$

$$\vec{\mathcal{V}}_{i+1}\ (x) = \min \left\{ \begin{array}{l} \vec{\mathcal{V}}_i\ (x), \\ \min\limits_{u} \max\limits_{v} (c(x, u, v) + \vec{\mathcal{V}}\ (f(x, u, v))) \end{array} \right\}$$

Rappelons que le choix des opérateurs $\max$ et de '+' dans cet équation est spécifique à ce critère de performance particulier. Quand max est remplacé par une somme pondérée, nous obtenons la procédure solution des processus de Markov contrôlés, conduisant à une stratégie avec une valeur prévue optimale. Quand l'addition est remplacée par l'opérateur $\max$, vous obtenez essentiellement l'algorithme de synthèse en arrière pour des systèmes à événements discrets (automates). Dans ce cas, la valeur de $\vec{\mathcal{V}}_i$ correspond à la fonction caractéristique de l'ensemble des états à partir desquels le contrôleur ne peut pas reporter l'atteinte d'un état interdit pour plus de i étapes.

Il est garanti que cette procédure élégante trouvera (si elle converge) la valeur optimale $\vec{\mathcal{V}}\ (x_0)$ du jeu ainsi que la stratégie qui atteint cet optimum : il suffit de prendre pour chaque x le u qui atteint l'optimum local. Pour les systèmes d'états finis avec des coûts de transition positifs, la convergence finie est garantie. Le seul inconvénient de la programmation dynamique est le besoin de calculer la fonction pour plus d'états que nécessaire, parfois des états qui ne peuvent pas être atteints du tout à partir de x_0, et parfois des états qui ne peuvent être atteints par aucune stratégie raisonnable. Cela empêche l'application directe de l'algorithme aux systèmes ayant un énorme espace d'états (la "malédiction de dimensionalité" de Bellman).

6. RECHERCHE EN AVANT

Les problèmes de plus court chemin (sans adversaire) admettent une procédure en avant duale, due à Dijks-

tra. Ici nous utilisons une fonction de valeur en arrière $\overleftarrow{\mathcal{V}}$ telle que $\overleftarrow{\mathcal{V}}\ (x)$ indique le coût minimal pour atteindre x à partir de x_0, et nous voulons calculer $\overleftarrow{\mathcal{V}}\ (x)$ pour $x \in F$. Cela est fait itérativement en utilisant :

$$\overleftarrow{\mathcal{V}}\ (x_0) = 0$$
$$\overleftarrow{\mathcal{V}}\ (x) = \min_u(c(x', u) + \overleftarrow{\mathcal{V}}\ (f(x', u)))$$

où x' prend ces valeurs parmi tous les prédécesseurs immédiats de x. Cet algorithme est polynômial aussi mais, comme nous l'avons vu, cela n'est pas d'une grande aide pour des graphes de transition exponentiels. L'avantage de cette procédure est la capacité à appliquer une recherche intelligente et de parfois trouver l'optimum sans explorer tous les états atteignables. Si on ne veut pas insister sur l'optimalité, on peut trouver des solutions raisonnables tout en explorant une petite fraction des chemins.

Pour démontrer cette idée d'une manière qui peut être étendue plus tard aux problèmes avec adversaires, nous considérons chaque chemin incomplet comme une stratégie partielle définie seulement sur les états rencontrés sur le chemin. Nous conserverons les triples de la forme $(s, x, \overleftarrow{\mathcal{V}})$ où s est une stratégie partielle, x correspond au dernier noeud sur le chemin et $\overleftarrow{\mathcal{V}}$ est le coût pour atteindre x en suivant le chemin. La première version de l'algorithme explore tous les chemins (et toutes les stratégies partielles). Nous utilisons une liste d'attente W dans laquelle nous conservons les chemins partiels qui doivent être explorés davantage. L'algorithme est donné ci-dessous :

$W := \{(\emptyset, x_0, 0)\}$
repeat
  Pick a non-terminal node $(s, x, \overleftarrow{\mathcal{V}}) \in W$
  for every $u \in U$ **do**
    $(s', x', \overleftarrow{\mathcal{V}}') :=$
      $(s \cup \{x \mapsto u\}, f(x, u), \overleftarrow{\mathcal{V}} + c(x, u))$
    Insert $(s', x', \overleftarrow{\mathcal{V}}')$ into W
  end
  Remove $(s, x, \overleftarrow{\mathcal{V}})$ from W
until W contains only terminal nodes

Si de nouveaux noeuds sont introduits à la fin de W, le graphe est exploré en largeur d'abord. Les noeuds explorés par cette procédure dans l'exemple de la figure 2 apparaissent dans la table ci-dessous. Les quatre entrées pour les noeuds finaux sont comparés et le chemin/stratégie à coût minimal est choisi.

s	x	$\overleftarrow{\mathcal{V}}$
$\emptyset$	x_0	0
$\{x_0 \mapsto u_1\}$	x_1	$c(x_0, u_1)$
$\{x_0 \mapsto u_2\}$	x_2	$c(x_0, u_2)$
$\{x_0 \mapsto u_1, x_1 \mapsto u_1, \}$	x_3	$c(x_0, u_1) + c(x_1, u_1)$
$\{x_0 \mapsto u_1, x_1 \mapsto u_2, \}$	x_3	$c(x_0, u_1) + c(x_1, u_2)$
$\{x_0 \mapsto u_2, x_2 \mapsto u_1, \}$	x_3	$c(x_0, u_2) + c(x_2, u_1)$
$\{x_0 \mapsto u_2, x_2 \mapsto u_2, \}$	x_3	$c(x_0, u_2) + c(x_2, u_2)$

Cet algorithme peut être modifié pour trouver une stratégie optimale sans exploration exhaustive. L'idée

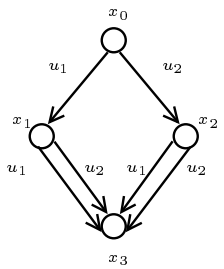

FIG. 2. Une problème de plus court chemin.

est d'associer à chaque stratégie partielle une fonction d'estimation qui donne une borne inférieure sur le coût de toute extension de la stratégie. Cette fonction est définie pour chaque $(s, x, \overleftarrow{\mathcal{V}})$ dans W comme $\mathcal{E}(s, x, \overleftarrow{\mathcal{V}}) = \overleftarrow{\mathcal{V}} + \underline{\vec{\mathcal{V}}}(x)$ où $\underline{\vec{\mathcal{V}}}$ est une sous-approximation de la fonction "cost-to-go" $\vec{\mathcal{V}}$. Cette fonction peut être dérivée à partir d'information spécifique au domaine et elle fournit une estimation optimiste du coût restant pour atteindre la cible par une stratégie qui étend s. Notez que quand x devient plus profond, le composant passé dans $\mathcal{E}(x)$ devient plus dominant et l'estimation - plus réaliste.

Une version "meilleur d'abord" de l'algorithme maintient W ordonné selon $\mathcal{E}$ et explore les noeuds les plus prometteurs d'abord. De plus l'algorithme peut arrêter d'explorer quand la valeur de $\mathcal{E}$ pour le premier élément dans W est plus grand que des solutions qui ont déjà été trouvées. Pour augmenter l'efficacité de l'algorithme nous pouvons d'abord effectuer une recherche en profondeur aléatoire pour obtenir quelques solutions aux premières étapes de la recherche ("branch and bound"). La recherche "meilleur d'abord" est garantie pour trouver l'optimum, et si nous relaxons les exigences d'optimalité, nous pouvons explorer encore moins d'états, par exemple, en explorant seulement un sous-ensemble de successeurs de chaque noeud, ou en arrêtant l'algorithme quand une solution plus petite qu'une certaine valeur prépécifiée est trouvée.

L'adaptation de la recherche en avant à des situations de graphes de jeu est réalisé comme suit. D'abord redéfinissions la fonction valeur comme :

$$\overleftarrow{\mathcal{V}}\ (x) = \min_u \max_v(c(x', u, v) + \overleftarrow{\mathcal{V}}\ (f(x', u, v))).$$

Ici, à cause de l'adversaire, chaque stratégie partielle résulte en un ensemble d'états. Pour éviter les notations lourdes nous omettons le coût des ensembles d'états atteignables et le lecteur doit garder à l'esprit que chaque atteignable x est, en fait, $(x, \overleftarrow{\mathcal{V}})$ où $\overleftarrow{\mathcal{V}}$ est le coût accumulé pour atteindre x via le chemin en question. L'ensemble des successeurs d'un état x via une action du contrôleur u est

$$f(x, u) = \{f(x, u, v) : v \in V\}.$$

Le u-successeur de (s, x) avec s étant une stratégie partielle est

$$\sigma((s, x), u)) = (s \cup \{x \mapsto u\}, f(x, u)).$$

Considérez maintenant un noeud de la forme (s, L) où s est une stratégie partielle et L est l'ensemble des états atteignables en suivant s. Les successeurs de (s, L), c'est-à-dire, les stratégies partielles qui étendent s et leurs ensembles respectifs d'états atteignables, sont tous des combinaisons de tous les choix possibles de u pour tout $x \in L$:

$$\sigma(s, L) = \bigotimes_{x \in L} \{(\sigma(s, x), u) : u \in U)\},$$

où

$$L_1 \otimes L_2 = \{(s_1 \cup s_2, m_1 \cup m_2) : (s_1, m_1) \in L_1, (s_2, m_2) \in L_2\}.$$

Illustrons cela sur l'arbre de jeu de la figure 1. Le noeud racine, $(\emptyset, \{x_0\})$, a deux successeurs :

$$\sigma(\emptyset, \{x_0\}) = \{\sigma((\emptyset, x_0), u_1), \sigma((\emptyset, x_0), u_2)\}$$

$$= \left\{ \begin{array}{l} (\{x_0 \mapsto u_1\}, \{x_1, x_2\}) \\ (\{x_0 \mapsto u_2\}, \{x_3, x_4\}) \end{array} \right\}$$

Calculons les successeurs du premier :

$$\sigma(\{x_0 \mapsto u_1\}, \{(x_1, x_2\}) =$$

$$\left\{ \begin{array}{l} (\{x_0 \mapsto u_1, x_1 \mapsto u_1\}, \{x_5, x_6\}), \\ (\{x_0 \mapsto u_1, x_1 \mapsto u_2\}, \{x_7, x_8\}) \end{array} \right\} \otimes$$

$$\left\{ \begin{array}{l} (\{x_0 \mapsto u_1, x_2 \mapsto u_1\}, \{x_9, x_{10}\}), \\ (\{x_0 \mapsto u_1, x_2 \mapsto u_2\}, \{x_{11}, x_{12}\}) \end{array} \right\}$$

ce qui donne les quatre noeuds suivants :

$$\left(\begin{array}{l} \{x_0 \mapsto u_1, x_1 \mapsto u_1, x_2 \mapsto u_1\}, \\ \{x_5, x_6, x_9, x_{10}\} \end{array} \right),$$

$$\left(\begin{array}{l} \{x_0 \mapsto u_1, x_1 \mapsto u_1, x_2 \mapsto u_2\}, \\ \{x_5, x_6, x_{11}, x_{12}\} \end{array} \right),$$

$$\left(\begin{array}{l} \{x_0 \mapsto u_1, x_1 \mapsto u_2, x_2 \mapsto u_1\}, \\ \{x_7, x_8, x_9, x_{10}\} \end{array} \right)$$

et

$$\left(\begin{array}{l} \{x_0 \mapsto u_1, x_1 \mapsto u_2, x_2 \mapsto u_2\}, \\ \{x_7, x_8, x_{11}, x_{12}\} \end{array} \right)$$

Une version exhaustive de l'algorithme de recherche en avant pour les graphes de jeu est exponentielle en la taille du graphe (contrairement à la procédure en arrière) mais avec l'aide d'une fonction d'estimation sur les ensembles de noeuds, on peut espérer éliminer de nombreuses branches de l'arbre de recherche en utilisant une recherche meilleur d'abord. Notez que la procédure peut être adaptée facilement au critère du cas moyen en remplaccant l'ensemble des états par les probabilités sur les états.

Le lecteur attentif peut avoir remarqué quelques inexactitudes dans la description spécifique aux arbres de jeu, plutôt qu'au cas plus général de graphes de jeu. Dans ce dernier cas, un ensemble d'états atteignables peut inclure plusieurs copies du même état, et toutes sauf celles avec le pire cas peut être enlevées. Nous avons aussi supposé qu'une stratégie ne peut pas atteindre le même état deux fois suivant le même chemin. Une stratégie qui fait cela a une valeur infinie et peut être rejetée. Pour la clarté d'exposition nous supposons que les deux joueurs peuvent appliquer n'importe quel élément de U et V à n'importe quel instant, tandis que dans la réalité certaines actions sont possibles seulement dans certains états.

Ceci conclu la première partie de l'article dans laquelle nous avons présenté trois approches pour résoudre des problèmes de contrôle optimal en la présence d'un adversaire. La première approche était basée sur l'optimisation en dimension finie et horizon borné. Les deux autres approches étaient basées sur la propagation de coûts suivant les chemins, soit en arrière (programmation dynamique) soit en avant. Ces approches ont été décrites en utilisant des U et V discrets et leur adaptation à des domaines continus n'est pas évidente, à moins qu'ils ne soient discrétisés. La discrètisation de U peut conduire à une perte d'optimalité et la discrétisation de V - à des valeurs optimistes de la stratégie choisie. Dans les sections suivantes nous démontrons cette approche sur un type intéressant de jeu joué avec des valeurs discrètes sur un temps continu ; pour être plus précis, l'ordonnancement avec tâches de durées incertaines.

7. L'ORDONNANCEMENT EN TANT QUE JEU

Les problèmes d'ordonnancement apparaissent dans diverses situations où l'utilisation de ressources au cours du temps doit être régulée. Un ordonnancer est un mécanisme qui décide à chaque instant d'allouer ou pas une ressource à l'une des tâches qui en a besoin. Malheureusement, la recherche d'ordonnancement s'étale sur plusieurs domaines d'application, et dans plusieurs d'entre eux des problèmes sont résolus en utilisant des méthodes spécifiques au domaine, sans conduire vers une théorie plus générale (sauf, peut-être, la recherche opérationnelle où l'ordonnancement est traité comme un problème d'optimisation statique, similaire à l'approche décrite en Section 4). Dans cette section nous allons reformuler l'ordonnancement dans notre terminologie de jeux dynamiques à deux joueurs.

D'un côté du problème nous avons les *ressources*, un ensemble $M = \{m_1, \ldots, m_k\}$ de "machines" dont nous supposons qu'elles sont fixées. De l'autre côté nous avons des *tâches*, des unités de travail qui requièrent l'allocation de certaines machines pendant certaines durées afin d'être accomplies. Dans un monde de ressources non bornées l'ordonnancement n'est pas un problème : chaque tâche choisit des ressources dès qu'elle en a besoin et s'achève dès que cela convient. Lorsque ce n'est pas le cas, deux tâches peuvent avoir besoin de la même ressource en même temps et l'ordonnancer doit résoudre ce conflit et décider à qui donner la ressource d'abord. Les tâches peuvent être liées entre elles par diverses conditions

d'interdépendance, dont la plus typique est la *précédence* : une tâche ne peut commencer qu'après que certaines autres tâches (ses prédécesseurs) soient terminées. Dans cet article nous supposons que l'ensemble des tâches est fixe et connu à l'avance.

Pour modéliser de telles situations en tant que jeux dynamiques nous devons d'abord fixer l'espace d'états. Pour nos objectifs nous prenons l'état du système à tout instant donné pour inclure les états des tâches (en attente, active, finie), le temps déjà écoulé (pour les tâches actives) et les états correspondants des machines (au repos, ou occupée quand elle est utilisée par une tâche active). Les actions de l'ordonnancer sont de deux types, le premier étant des actions de la forme *start(p)* qui signifie allouer une machine m à la tâche p pour qu'elle puisse s'exécuter. L'effet d'une telle action sur un état où p est prêt a exécuté (tous ses prédécesseurs sont achevés) et m est inactive, est de rendre p active et m occupée. Notons cet ensemble d'actions S. L'autre "action" de l'ordonnancer est de ne rien faire, notée $\perp$. Dans ce cas les tâches actives continuent d'exécuter, les tâches en attente continuent d'attendre et le temps s'écoule. Les actions de l'environnement consistent en un ensemble d'actions de la forme $end(p)$ dont l'effet, lorsque la tâche a passé suffisamment de temps en état actif, est de faire passer la tâche à un état final et libérer la machine. Nous supposons que l'environnement est déterministe, c'est-à-dire, chaque transition $end(p)$ a lieu exactement d fois après le $start(p)$ où d est la durée pré-spécifiée de la tâche (par la suite, nous relâcherons cette hypothèse). Dans ce cas la stratégie peut être vue comme un unique ordonnancement, une fonction $s : \mathbb{R}_+ \to S \cup \{\perp\}$. Pour tout sauf un nombre fini d'instances de temps nous avons $s(t) = \perp$ et l'ordonnancement est déterminé par un nombre finie de dates spécifiant le debut d'activation pour chaque tâche.

8. ORDONNACEMENT DE "JOB SHOP" DETERMINISTE

Un problème de "job shop" consiste en un ensemble fini $J = \{J^1, \ldots, J^n\}$ de travaux ("job") qui doivent être traités sur un ensemble M de machines. Chaque job J^i consiste en une séquence finie de tâches qui devrait être exécutées l'une après l'autre, où chaque tâche est caractérisée par un couple de la forme (m, d) avec $m \in M$ et $d \in \mathbb{N}$, indiquant l'utilisation nécessaire de machine m pour une durée de temps fixée d. Chaque machine peut traiter au plus une tâche simultanément et, à cause des contraintes de précédence, une tâche au plus de chaque job peut être actif à tout instant. Les tâches ne peuvent pas être préempt'e une fois commencée. Nous voulons déterminer les temps de début pour chaque tâche de manière à ce que le temps total d'exécution de tous les jobs (le temps où la dernière tâche s'achève) soit minimal.

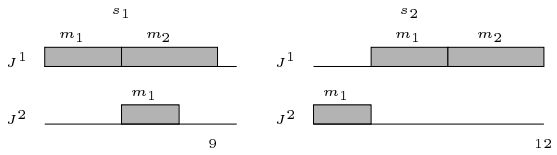

FIG. 3. Deux ordonnancement s_1 and s_2 pour l'exemple.

En tant qu'exemple considérons le problème

$$J^1 : (m_1, 4), (m_2, 5) \;\; J^2 : (m_1, 3)$$

pour lequel il y as un conflit sur m_1. Ce conflit peut être résolu de deux manières, soit en donnant la priorité à J^1 soit à J^2 (ordonnancements s_1 et s_2 de la Figure 3). La longueur induite par s_1 est 9 et il est l'ordonnancement optimal pour cet exemple. La difficulté du problème vient du fait que parfois un ordonnancement optimal est atteint en n'exécutant pas une tâche dès qu'elle est prête afin de garder une machine libre pour une autre tâche dont on aura besoin à l'avenir.

La manière traditionnelle de résoudre ce problème est d'assigner des variables, z_1, z_2 et z_3 aux temps de début des trois tâches, une variable z_4 à la longueur totale de l'ordonnancement et de résoudre un problème d'optimisation sous contraintes. Les contraintes de précédence pour J^1 sont exprimées par $z_2 \geq z_1 + 4$ (elle ne peut pas utiliser m_2 avant de finir m_1). Le fait qu'une seule tâche peut utiliser m_1' à un moment donné est exprimé par la condition

$$[z_1, z_1 + 4] \cap [z_2, z_2 + 3] = \emptyset$$

déclarant que les périodes d'utilisation de m_1 par les deux jobs ne doivent pas coïncider. Tout le problème est ainsi formulé par :

$$\begin{aligned} &\min(z_4) \;\; \text{subject to} \\ &z_2 - z_1 \geq 4 \\ &z_4 - z_2 \geq 5 \\ &z_4 - z_3 \geq 3 \\ &(z_2 - z_1 \geq 4 \vee z_1 - z_2 \geq 3) \end{aligned}$$

Le format de ce problème est en même temps plus simple et plus complexe que la programmation linéaire générale. D'un côté les contraintes sont toujours de la forme $z_i - z_j \geq d$ plutôt que des inégalités linéaires arbitraires. De l'autre côté, la dernière contrainte disjonctive, qui exprime un choix discret, rend l'ensemble de solutions faisables non-convexe. Quand le problème devient plus grand, l'ensemble de solutions faisables devient de plus en plus fragmenté en une union disjointe de polyèdres convexes dont le nombre est exponentiel en le nombre de conflits. Comme beaucoup d'autres problèmes d'optimisation combinatoire, le problème de job shop est NP-difficile et cela suggère que tout algorithme pourrait, dans certains cas, finir par énumérer toutes les solutions possibles.

Il mérite d'être mentionné que les gens habitués à l'optimisation continue tendent à transformer le pro-

blème en un problème de programmation linéaire en variables mixtes en introduisant des variables entières avec lesquelles il est possible de coder des disjonctions sous forme de contraintes arithmétiques. Le problème est alors transformé via la relaxation (en supposant temporairement que ces variables sont des valeurs réelles) en un programme convexe linéaire qui peut être résolu efficacement. Il reste alors à transformer la "solution" obtenue en une solution faisable avec des valeurs entières à la place des variables relaxées. Bien que cette approche fonctionne bien pour certaines classes de problèmes, j'ai des doutes concernant son utilité pour l'ordonnancement, un problème dominé par des choix discrets n'ayant pas d'interprétation numérique.

9. ORDONNANCEMENT AVEC AUTOMATES TEMPORISÉS

Dans ce chapitre je décrit plus en détail la modélisation de situations d'ordonnancement en tant que système dynamique sur lequel des chemins optimaux et des stratégies optimales peuvent être calculé en utilisant l'algorithme de recherche en avant décrit dans la section 6 Nous utilisons le modèle d'automate temporisé qui s'est établi en tant que formalisme de choix pour décrire des comportements discrets dépendants du temps. Les automates temporisés sont des automates opérant* dans le domaine du temps dense. Leur espace d'états est le produit d'un ensemble fini d'états discrets et de l'espace des horloge $\mathbb{R}_+^m$ (ensemble de valeurs des variables des horloges). Le comportement de l'automate consiste en une alternance de période de passage du temps où l'automate reste au même lieu et les valeurs des horloges grandissent uniformément, et de transitions instantanées qui peuvent être prises quand les valeurs des horloges satisfont certaines conditions et qui peuvent ramener certaines horloges à zéro. L'interaction entre les valeurs des horloges et les transitions discrètes est spécifiée par des conditions sur l'espace des horloge qui déterminent quelle évolution future, un passage du temps ou franchissement d'une ou plusieurs transitions, est possible à cet état.

Quand des automates temporisés modélisent des problèmes d'ordonnancement, les états discrets enregistrent l'état qualitatif du problème d'ordonnancement (qui s'exécute, qui est terminé) et les horloges donnent la composante quantitative de l'état, autrement dit le temps que chaque tâche active a déjà passé à s'exécuter. Nous supposons ici qu'il y a une seule machine de chaque type et qu'ainsi les états des machines sont impliqués par les états des tâches. Nous épargnerons au lecteur la définition formelle exacte des automates à temporisés et illustrerons notre approche de modélisation via un exemple.

Nous commenccons par modéliser chaque job comme un automate simple avec une horloge. Les automates

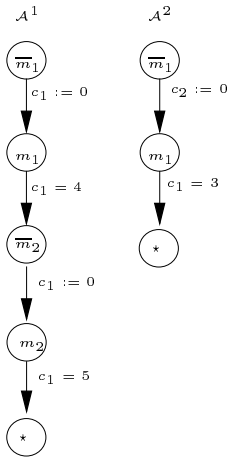

FIG. 4. Automates pour les deux jobs.

pour notre exemple sont représentés dans la figure 4. L'automate $\mathcal{A}_1$ commence avec l'état $\overline{m}_1$ où il attend la machine m_1. Il reste dans cet état jusqu'à ce qu'une transition pour activer l'état m_1 soit prise. Cette transition "start" est fournie par l'ordonnancer et elle remet l'horloge c_1 à zéro. L'automate reste à cet état jusqu'à ce que l'horloge atteigne 4 et passe alors à l'état $\overline{m}_2$, attendant la prochaine tâche et ainsi de suite jusqu'à atteindre l'état final. Les transitions "end" partant d'états actifs sont réalisées par l'environnement et sont considérées comme des actions non contrôlées par l'ordonnancer. Les horloges sont considérées "inactives" aux états d'attente car elles sont remises à zéro avant d'être testées.

Ces automates décrivent les comportements possibles de chaque job de faccon indépendant. Leur comportement joint sous contraintes de ressources est modélisé par leur produit montré à la figure 5. C'est essentiellement un produit cartésien des automates de job, où les contraintes de ressources sont exprimées en enlevant des états tels que (m_1, m_1) où plus d'un job utilise la machine. Cela produit un "trou" dans l'automate et l'ordonnancer doit décider comment contourner ce trou, en donnant la machine à J^1 ou à J^2. Les deux ordonnancements de la figure 3 correspondent aux deux comportements (chemins, éxecutions) suivants de l'automate (nous utilisons la notation $\perp$ pour indiquer des horloges inactives, et $\xrightarrow{0}$ pour des actions discrètes telles que commencer ou finir une tâche) :

s_1 :
$(\overline{m}_1, \overline{m}_1, \perp, \perp) \xrightarrow{0} (m_1, \overline{m}_1, 0, \perp) \xrightarrow{4} (m_1, \overline{m}_1, 4, \perp) \xrightarrow{0}$
$(\overline{m}_2, \overline{m}_1, \perp, \perp) \xrightarrow{0} (\overline{m}_2, m_1, 0, \perp) \xrightarrow{0} (m_2, m_1, 0, 0) \xrightarrow{3}$
$(m_2, m_1, 3, 3) \xrightarrow{0} (m_2, \star, 3, \perp) \xrightarrow{2} (m_2, \star, 5, \perp) \xrightarrow{0}$
$(\star, \star, \perp, \perp)$

s_2 :
$(\overline{m}_1, \overline{m}_1, \perp, \perp) \xrightarrow{0} (\overline{m}_1, m_1, \perp, 0) \xrightarrow{3} (\overline{m}_1, m_1, \perp, 3) \xrightarrow{0}$
$(\overline{m}_1, \star, \perp, \perp) \xrightarrow{0} (m_1, \star, 0, \perp) \xrightarrow{4} (m_1, \star, 4, \perp) \xrightarrow{0}$
$(\overline{m}_2, \star, \perp, \perp) \xrightarrow{0} (m_2, \star, 0, \perp) \xrightarrow{5} (m_2, \star, 5, \perp) \xrightarrow{0}$
$(\star, \star, \perp, \perp)$

Il n'est pas difficile de voir la correspondance entre l'ensemble des comportements possibles de l'automate qui atteignent l'état final et l'ensemble de tous

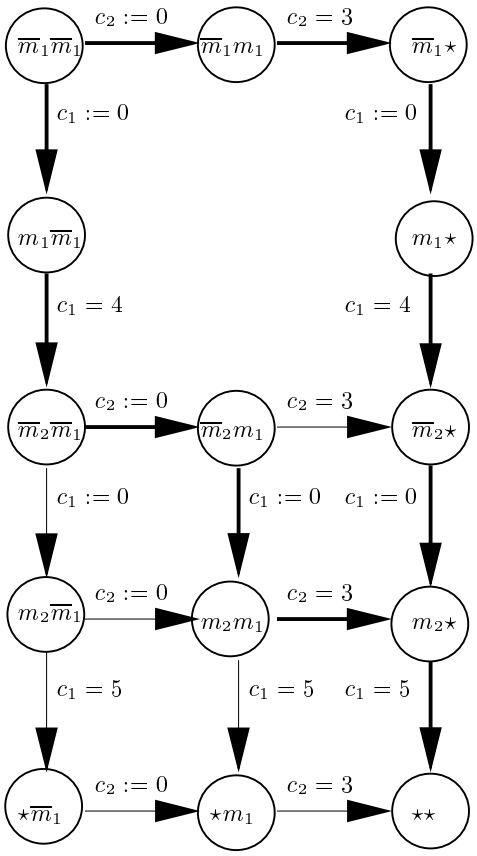

FIG. 5. L'automate global pour les deux jobs. Les chemins qui correspondent aux deux ordonnancements sont indiqués par les traits gras.

les ordonnancements faisables. Ainsi le problème d'ordonnancement optimal est réduit à trouver le chemin le plus court dans un automate temporisé, où la longueur du chemin est le temps total écoulé. Le nombre de tels chemins est non-dénombrable (chaque automate peut rester n'importe quelle quantité de temps dans un état d'attente) cependant nous avons montré que l'optimum est trouvé parmi un nombre fini de chemins et chaque noeud dans l'arbre de recherche a un nombre fini de successeurs méritant d'être explorés. Le nombre de tels chemins est encore exponentiel et une recherche exhaustive est irréalisable. Notre implantation d'un algorithme de recherche meilleur d'abord sur ce modèle pourrait trouver des ordonnancements optimaux pour des problèmes avec 6 jobs 6 machines et 36 tâches. Au-delà de cca nous devons appliquer une heuristique capable à trouver des solutions 5% de l'optimum connu pour des problèmes allant jusqu'à 15 jobs, 15 machines et 225 tâches.

Le lecteur a probablement remarqué que le modèle dynamique utilisé ici ne rentre pas exactement dans le cadre synchrone de temps discret décrit précédemment. En utilisant une approche "échantillonnée" et restreignant les événements à avoir lieu et à être observé seulement à des multiples d'une certaine constante δ, nous pouvons approximer tout automate temporisé par un système à temps discret. Cependant, quand des événements ont lieu de manière clairsemée au cours du temps, l'approche asynchrone à temps continu est plus efficace car elle permet d'accélérer l'évolution du système en laissant le temps avancer jusqu'à l'événement suivant.

10. ORDONNACEMENT SOUS INCERTITUDE

Bien que l'approche qui vient d'être décrite soit élégante, on pourrait affirmer que le monde de l'ordonnancement pourrait vivre sans une technique de plus pour résoudre le problème de job shop. L'avantage qu'il y a à utiliser des modèles dynamiques basés sur des états se manifeste quand nous passons aux problèmes plus complexes d'ordonnancement sous incertitude. La recherche académique en ordonnancement a souvent été critiquée d'un point de vue pratique en cause de suppositions irréalistes et il a été noté que des vrais ordonnancements sont rarement exécutés comme prévu. Au cours de l'exécution il peut arriver que des tâches se terminent plus tôt ou plus tard que prévu, de nouvelles tâches peuvent apparaître, des machines peuvent cesser de fonctionner, etc. Dans de telles situations, ce qu'il nous faut est une politique d'ordonnancement, une stratégie qui s'adapte à l'évolution de système a contrôler et modifie ses décisions en conséquence. Dans cette section nous augmentons le problème de job shop avec un type d'incertitude, plus précisément l'incertitude liée à la des durée des tâches. Cela signifie qu'une description de tâche prend la forme $(m, [l, h])$ indiquant que la véritable durée de la tâche est un certain $d \in [l, h]$. Chaque instance du problème de job shop consiste à choisir un tel d pour chaque intervalle et nous devons évaluer une stratégie d'après sa performance sur toutes ces instances. Considérez le problème

$$J^1 : (m_1, 10), (m_3, [2, 4]), (m_4, 5)\ J^2 : (m_2, [2, 8]), (m_3, 7)$$

où la seule ressource en conflit est m_3 et l'ordre de son utilisation est la seule décision du contrôleur. Les incertitudes concernent les durées de la première tâche de J^2 et de la seconde tâche de J^1. Ainsi une instance est un couple $d = (d_1, d_2) \in [2, 4] \times [2, 8]$. La Figure 6-(a) représente les ordonnancements optimaux pour les instances $(8, 4)$, $(8, 2)$ et $(4, 4)$ qui auraient pu être trouvés par un ordonnancer non-causal clairvoyant qui connaît toute l'instance à l'avance. Mais les instances se révèlent progressivement au cours de l'exécution : la valeur de d_1, par exemple, est connue seulement après la fin de la seconde tâche de J^1.

Il se trouve que pour ce type particulier d'incertitude, l'optimisation par rapport au critère du pire scénario est quelque peu triviale. Il y a toujours une instance maximale (critique), $(8, 4)$ dans cet exemple, ayant deux propriétés importantes : 1) l'ordonnancement optimal pour cette instance est valide aussi pour toutes

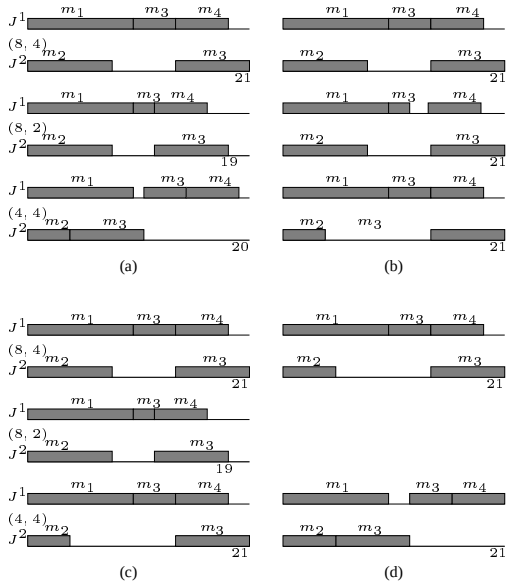

FIG. 6. (a) Optimal schedules for three instances ; (b) A static schedule based on the worst instance $(8, 4)$; (c) The behavior of a hole filling strategy based on instance $(8, 4)$; (d) The equal performance of the two strategies on instance $(5, 4)$.

les autres instances plus petites (ignorer la résiliation précipité de certaines tâches et garder la machine occupée jusqu'à ce qu'un temps h s'écoule) ; 2) aucune stratégie ne peut mieux répondre à cette instance. La figure 6-(b)) montre le comportement d'une stratégie statique du pire scénario basée sur l'instance $(8, 4)$ et on peut voir que c'est un gaspillage pour d'autres instances. Nous voulons un ordonnancer adaptatif plus intelligent qui prend la véritable durée de m_2 en considération.

L'une des manières les plus simples d'être adaptatif est la suivante. D'abord nous choisissons une *instance nominale* d et trouvons un ordonnancement s qui est optimal pour cette instance. Plutôt que de prendre s "littéralement" comme une assignation de date de demarage absolu au chaque tâches, nous en extrayons seulement *l'information qualitative*, l'ordre dans lequel les tâches en conflit utilisent chaque ressource. Dans notre exemple l'ordonnancement optimal pour l'instance $(8, 4)$ est associé au fait de donner la priorité à J^1 sur m_3. Alors, au cours de l'exécution, nous commenccons chaque tâche dès que ses prédécesseurs se sont terminés, pourvu que l'ordre soit respecté. Comme le montre la Figure 6-(c), une telle stratégie est meilleure que l'ordonnancement statique pour des instances telles que $(8, 2)$ elle profite de la résiliation précipité de la seconde tâche de J^1 et "décale en avant" les temps de début des tâches qui suivent.

Notez que cette stratégie de "boucher de trous" n'est pas limitée au pire cas. On peut utiliser toute instance nominale puis faire passer des tâches en avant ou en arrière dans le temps selon le besoin tout en maintenant l'ordre. D'un autre côté, un ordonnancement statique ne peut être basé que sur le scénario du pire - un ordonnancer basé sur une autre instance nominale pourrait supposer qu'une ressource est accessible à un moment donné, alors qu'en réalité elle sera occupée.

La stratégie de remplissage de trous est optimale pour toutes les instances dont l'ordonnancement optimal a le même ordre que celui de l'instance nominale. Elle n'est pas bonne, cependant, pour des instances telles que $(4, 4)$ qui ne peuvent pas bénéficier de la fin précoce de m_2 car faire passer m_3 de J^2 en avant violerait la priorité sur m_3. Pour de tels cas il faut une forme plus raffinée d'adaptativité. A regarder les ordonnancements optimaux pour $(8, 4)$ et $(4, 4)$ dans la Figure 6-(a), nous observons que dans les deux la décision de donner m_3 à J^2 ou pas est prise au même état qualitatif où m_1 s'exécute et m_2 s'est achevé. La seule différence réside dans le temps d'exécution écoulé de m_1 au point de décision. Ainsi un ordonnancer adaptatif devrait baser ses décisions également sur l'information quantitative encodée dans les valeurs d'horloge.

Considérons l'approche suivante : initialement nous trouvons un ordonnancement optimal pour une instance nominale. Pendant l'exécution, chaque fois qu'une tâche s'achève nous réordonnanccons le problème "résiduel", supposant des durées nominales pour des tâches qui ne se sont pas encore achevées. Dans notre exemple, nous construisons d'abord un ordonnancement optimal pour $(8, 4)$ et commenccons à l'exécuter. Si la tâche m_2 dans J^2 se termine après 4 unités de temps nous obtenons le problème résiduel

$$J'_1 : (\mathbf{m_1}, \mathbf{6}), (m_3, 4), (m_4, 5) \qquad J'_2 : (m_3, 7)$$

où les lettres en gras indiquent que m_1 doit être exécutée immédiatement (elle s'exécute déjà et nous supposons qu'il n'y a aucune préemption). Pour ce problème la solution optimale sera de donner m_3 à J^2. De même, si m_2 se termine à 8 nous avons

$$J'_1 : (\mathbf{m_1}, \mathbf{2}), (m_3, 4), (m_4, 5) \qquad J'_2 : (m_3, 7)$$

et l'ordonnancement optimal consiste à attendre la fin de m_1 pour donner m_3 à J^1. La propriété des ordonnancements ainsi obtenus est qu'ils sont optimaux par rapport à l'hypothèse nominale concernant le futur à tout état atteignable durant l'exécution. Nous appelons de telles stratégies *optimales d-futur*. C'est le principe sous-jacent du contrôle prédictif de modèle où à chaque étapes, les actions à l'état "réel" courant sont ré-optimisées en supposant une prédiction nominale pour un horizon de futur borné. Un défaut majeur de cette approche est qu'elle implique beaucoup de calcul "en ligne", résolvant un nouveau problème d'ordonnancement chaque fois qu'une tâche s'achève. Cela limite son applicabilité aux processus "lents". Dans la prochaine section nous présentons une approche alternative où une stratégie équivalente est synthétisée hors ligne en utilisant une variante symbolique de la programmation dynamique adaptée aux automates temporisés.

11. PROGRAMMATION DYNAMIQUE SUR AUTOMATES TEMPORISÉS

L'espace d'états d'un automate temporisé consiste en des paires de la forme (q, c) où $q = (q_1, \ldots, q_n)$ est un état discret, indiquant les états locaux de tous les jobs, et $c = (c_1, \ldots, c_n)$ est un vecteur de valuations d'horloges appartenant à un sous-ensemble borné des réels non négatifs. On y trouve une fonction valeur $\vec{\mathcal{V}}$ telle que $\vec{\mathcal{V}}\ (q, c)$ dénote le temps minimum pour atteindre l'état final à partir de (q, c), en supposant des valeurs nominales pour les tâches qui ne se sont pas terminées. Avant de donner la définition formelle donnons une explication intuitive. Étant à (q, c), tous les choix locaux de l'ordonnancer peuvent être ramenés à la forme suivante : laisser un temps t passer et alors exécuter une transition qui est rendue disponible par les valeurs d'horloge. Cette définition couvre aussi la possibilité d'une action immédiate ($t = 0$), ainsi que la possibilité d'attendre jusqu'à ce qu'une transition incontrôlée soit prise par l'environnement. La valeur induite par ce choix est la somme du temps d'attente t et de la valeur de l'état atteint après la transition. Cela ce résume par la définition récursive suivante :

$$\vec{\mathcal{V}}\ (\star, c) = 0$$
$$\vec{\mathcal{V}}\ (q, c) =$$
$$\min\{t + \vec{\mathcal{V}}\ (q', c') : (q, c) \xrightarrow{t} (q, c + t\mathbf{1}) \xrightarrow{0} (q', c')\}$$

Pour illustrer le calcul de $\vec{\mathcal{V}}$ nous considérons une version simplifiée de l'exemple de la section précédente avec une seule durée incertaine :

$$J^1 : (m_1, 10), (m_3, 4), (m_4, 5)\ \ J^2 : (m_2, [2, 8]), (m_3, 7).$$

La figure 7 montre la partie finale de l'automate global correspondant au problème, qui inclut l'état $(m_1, \overline{m}_3)$ où une décision de l'ordonnancer doit être prise. Le calcul commence avec $\vec{\mathcal{V}}\ (\star, \star, \bot, \bot) = 0$. La valeur de $(m_4, \star, c_1, \bot)$ est le temps qu'il faut pour satisfaire la condition $c_1 = 5$, qui est $5 \dot{-} c_1$. De même pour $\vec{\mathcal{V}}\ (\star, m_3, \bot, c_2) = 7 \dot{-} c_2$. Dans l'état (m_4, m_3) les deux jobs sont actifs et la transition à prendre dépend de celui qui "gagne la course" et termine d'abord :

$$\vec{\mathcal{V}}\ (m_4, m_3, c_1, c_2)$$

$$= \min \left\{ \begin{array}{l} 7 \dot{-} c_2 + \vec{\mathcal{V}}\ (m_4, \star, c_1 + (7 \dot{-} c_2), \bot), \\ 5 \dot{-} c_1 + \vec{\mathcal{V}}\ (\star, m_3, \bot, c_2 + (5 \dot{-} c_1)) \end{array} \right\}$$

$$= \min\{5 \dot{-} c_1, 7 \dot{-} c_2\}$$

$$= \left\{ \begin{array}{l} 5 \dot{-} c_1 \text{ if } c_2 \dot{-} c_1 \geq 2 \\ 7 \dot{-} c_2 \text{ if } c_2 \dot{-} c_1 \leq 2 \end{array} \right.$$

Notez que les deux transitions sont des transitions de fin incontrôlées et qu'aucune décision de l'ordonnancer n'est requise à cet état. Le calcul procède en arrière, calculant $\vec{\mathcal{V}}$ pour tous les états. En particulier, pour l'état $(m_1, \overline{m}_3)$ où nous devons choisir entre

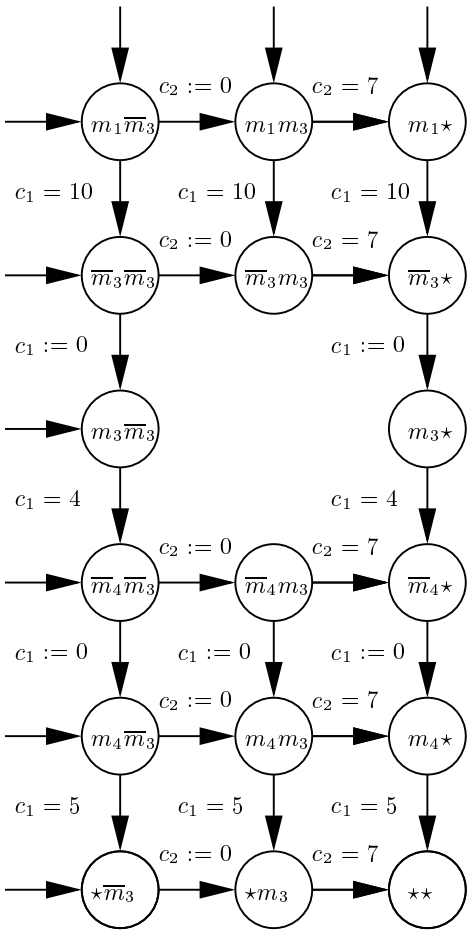

FIG. 7. Une partie d'automat global.

donner m_3 immédiatement à J^2 ou attendre la fin de m_1 pour donner m_3 à J^1, nous obtenons :

$$\vec{\mathcal{V}}\ (m_1, \overline{m}_3, c_1, \bot) = \min\{16, 21 \dot{-} c_1\}$$

$$= \left\{ \begin{array}{ll} 16 & \text{if } c_1 \leq 5 \\ 21 \dot{-} c_1 & \text{if } c_1 \geq 5 \end{array} \right.$$

Ainsi, si m_1 se termine après moins de 5 unités de temps il vaut mieux donner m_3 à J^2, sinon il vaut mieux attendre et la donner à J^1. La Figure 6-(d) montre que, en effet, les deux stratégies coïncident en performance quand $c_1 = 5$.

Le lecteur ne doit pas se laisser abuser par le succès de notre stratégie à atteindre la performance d'un ordonnancer clairvoyant dans ce petit exemple. Dans des problèmes légèrement plus complexes avec plusieurs incertitudes il est impossible de concurrencer avec la connaissance du futur et être optimal d-futur est suffisant.

Le calcul en soit de la fonction valeur est implanté en utilisant des techniques d'atteignabilité standards pour des automates temporisés qui sont en dehors du champ de l'article présent. Nous avons testé notre implantation sur un problème avec 4 jobs, 6 machines et 24 tâches, dont 8 ayant des durées incertaines. Nous avons fixé deux instances, une "optimiste" où chaque durée de tâche est fixée à l, et une "pessimis-

te" avec h durées. Nous avons appliqué notre algorithme pour trouver deux stratégies optimales d-futur et deux stratégies de remplissage de trous basées sur ces instances. Nous avons généré 100 instances aléatoires avec des durées récupérées uniformément dans chaque intervalle $[l, h]$, et comparé les résultats des stratégies mentionnées plus haut avec un ordonnancer clairvoyant optimal qui connaît chaque d à l'avance, et avec l'ordonnacer statique du pire cas. Il ressort que l'ordonnancement statique est, en moyenne, plus long que l'optimum de $12,54\%$. La stratégie de remplissage de trous dévie de l'optimum de $4,90\%$ (pour des prédictions optimistes) et $4,44\%$ (pour des prédictions pessimistes). Notre stratégie produit des ordonnancements qui sont plus long que l'optimum de $1,40\%$ et $1,14\%$, respectivement. La bonne nouvelle est que notre stratégie est bien meilleure que l'ordonnancement statique, et peut être considérée comme un bon outil pour des systèmes avec des critères de performance en temps réel "mou". La mauvaise nouvelle est qu'elle est beaucoup plus coûteuse que la stratégie de remplissage de trous. Cette dernière résout un problème sans adversaire et peut utiliser une recherche en avant intelligente tandis que le calcul de notre stratégie doit explorer tout l'espace d'état. L'adaptation d'une recherche en avant sur les arbres de jeu à ce problème n'est pas évident, à cause de la densité de l'ensemble d'actions de l'adversaire, et elle est sujette à des recherches en cours bien que l'adaptation de cette approche à d'autres types d'incertitude tels que des temps d'arrivé imprécis ou une incertitude discrète associée à des dépendances conditionnelles entre les tâches.

12. DISCUSSION

Les gens qui sont experts dans leur domaine sont souvent sceptiques envers des propositions de théories unifiées. En effet, comparées aux succès de la recherche de domaine spécifique, divers tendances holistiques telles que la "théorie générale de systèmes" se sont révélées par le passé comme étant plutôt stériles. Dire que "tout est des systèmes" et que plusieurs choses qui ont l'air si différentes sont, à un certain niveau d'abstraction, similaires, ne résout pas nécessairement des problèmes. J'espère que le cadre présenté dans cet article subira un meilleur sort. Il est moins ambitieux que certains de ses prédécesseurs dans le sens où il n'essaie pas de prédire l'imprévisible et faire semblant de donner des recettes optimales pour des phénomènes socio-économiques ou biologiques complexes pour lesquels nous ne connaissons même pas le vocabulaire de modélisation approprié. Il est restreint à des situations où des modèles dynamiques utiles et des critères de performance existent, des modèles qui sont déjà utilisées, implicitement ou explicitement, pour la simulation, vérification ou optimisation. Ce cadre est orienté vers un but concret : développer un outil pour définir et résoudre des problèmes de contrôle optimal pour des systèmes avec divers types de dynamiques.

Certains principes sous-jacents d'un tel cadre (dont certains existent déjà dans des domaines respectifs) sont mentionnés plus bas. D'abord, je crois que des systèmes doivent être définis avec une sémantique claire à partir de laquelle il est facile de voir qui sont les joueurs, quelles sont les variables qu'ils peuvent observer et influencer, qu'est-ce qui constitue un comportement du système, qu'est-ce qui EST supposé concernant l'environnement et quels sont les critères de performance naturels. A ce niveau, la description devrait être séparée des techniques de calcul spécifiques qui sont utilisées pour raisonner sur le modèle. Ceci est en contraste avec les approches à domaine spécifique où les problèmes sont souvent énoncés en termes biaisés envers des techniques de solution particulières et, parfois, accidentelles qui sont communes dans le domaine.

Après qu'un contrôleur optimal idéal ait été mathématiquement défini, les problèmes algorithmiques doivent être adressés. Ici la différence entre les classes de dynamiques de systèmes est manifestée par le type de problème d'optimisation sous contraintes à résoudre, discret (logique), continu (numérique) ou hybride. Dans la plupart des cas l'optimalité globale de la solution est une affaire cérémonial. Personne ne compte vraiment être optimal et les modèles sont de toute faccon imprécis. Dans certains cas, prouver une relation entre des solutions approximatives et l'optimum est une bonne mesure de la qualité d'une technique, mais ce n'est ni une condition nécessaire ni suffisante pour son utilité.

Puisqu'il reste de l'espace, permettez-moi d'ajouter quelques remarques polémiques. Il me semble que dans de nombreux domaines ayant rapport à cet article, il y a une tension entre les approches mathématique (théorique) et d'ingénierie ("hacking"). Le (vrai) pratiquant ne peut pas choisir les problèmes qu'il doit résoudre et n'a pas non plus le temps de développer de belles théories. Dans plusieurs cas il adaptera des solutions fournies par les mathématiciens de précédentes générations pour faire le travail. Le théoricien est supposé avoir l'esprit plus ouvert et explorer de nouvelles classes de modèles pour de nouveaux phénomènes mais la structure du monde académique ne l'encourage pas toujours à faire ainsi. Les membres des communautés scientifiques s'imposent souvent des critères d'évaluation intrinsèques qui dévient avec le temps de la raison d'être du domaine. Il n'y a aucun mal aux (bonnes) mathématiques pour les mathématiques en soit, mais il ne faut pas les confondre avec la résolution de vrais problèmes d'ingénierie ou même avec le fait de poser les fondations pour des solutions futures. Ce qui est vraiment nécessaire est un milieu entre la mathématique et l'ingénierie, qui nous permettent de voir les objets mathématiques génériques derrières les instances d'ingénierie, avec un

sens critique fort envers les traditions des champs académiques respectifs, qui sont souvent des effets de bord de la sociologie des communautés scientifiques plutôt que le résultat d'une véritable tentative d'être pertinent.

Remerciements : Toute tentative de couvrir de manière juste tous les travaux liés, sur une durée de plus de 50 ans dans de nombreuses communautés scientifiques, sera prétentieux ou nécessitera un effort allant au-delà de l'envergure de cet article. En conséquence, je ne fournis pas de références pour la première partie de l'article. Les résultats concernant l'ordonnancement ont été obtenus en collaboration avec Y. Abdeddaïm et E. Asarin. Les lecteurs intéressés par plus de détails devrait voir la thèse [A02] ou l'article [AAM04].

Cet article a été inspiré en partie par une interaction avec des automaticiens à diverses occasions liées aux systèmes hybrides (discrets/continus), y compris mes partenaires au projet CC, que j'aimerai remercier (les opinions sont, bien entendu, les miennes). Le travail sur l'ordonnancement a profité de relations similaires avec des partenaires dans le projet AMETIST. Ce manuscrit a bénéficié de commentaires fournis par Anil Nerode, Anders Ravn, Tariq Samad, Michel Sintzoff et Pravin Varaiya, et de nombreuses discussions passées avec Eugène Asarin et Bruce Krogh.

Références

[A02] Y. Abdeddaïm, *Scheduling with Timed Automata*. PhD thesis, Institut National Polytechnique de Grenoble, Laboratoire Verimag, 2002.

[AAM04] Y. Abdeddaïm, E. Asarin and O. Maler, Scheduling with Timed Automata, *Theoretical Computer Science*, 2004.