UNIVERSITÉ de CAEN/BASSE-NORMANDIE

U.F.R. : Sciences

ÉCOLE DOCTORALE : SIMEM

THÈSE

présentée par

Simon Le Gloannec

et soutenue

le 29 juin 2007

en vue de l'obtention du

DOCTORAT de l'UNIVERSITÉ de CAEN

spécialité : Informatique

(Arrêté du 7 août 2006)

Contrôle adaptatif d'un agent rationnel à ressources limitées dans un environnement dynamique et incertain

MEMBRES du JURY

Rachid Alami	Directeur de recherche	LAAS Toulouse CNRS	(rapporteur)
Philippe Preux	Professeur	Université Lille 3	(rapporteur)
Olivier Sigaud	Professeur	Université Paris VI	
Shlomo Zilberstein	Professeur	University of Massachusetts Amherst	
Abdel-Illah Mouaddib	Professeur	Université de Caen	(directeur)
François Charpillet	Directeur de recherche	LORIA Lorraine	(directeur)

Mis en page avec la classe thloria.

Remerciements

Je tiens à adresser toute ma reconnaissance au professeur Abdel Illah Mouaddib. C'est grâce à sa patience, sa disponibilité, son optimisme et ses conseils avisés que j'ai pu effectuer ma thèse dans d'aussi bonnes conditions.

Je témoigne toute ma gratitude à François Charpillet pour m'avoir accueilli au sein de l'équipe MAIA du LORIA. A Nancy, j'ai découvert une équipe dynamique et passionnée par les systèmes intelligents et la robotique. J'ai toujours apprécié d'avoir deux avis complémentaires et constructifs sur mes recherches.

Je remercie monsieur Rachid Alami, Directeur de Recherche au LAAS, de me faire l'honneur de participer à ce jury de thèse. Je remercie le professeur Philippe Preux d'avoir porté un jugement critique de qualité sur la forme finale de mon travail. Je remercie enfin les professeurs Olivier Sigaud et Shlomo Zilberstein pour l'intérêt qu'ils portent à mon travail.

Un merci tout spécial à l'homme qui murmurait à l'oreille des Koalas, Laurent Jeanpierre.

Je tiens ensuite à saluer les membres de l'équipe MAIA pour l'accueil qui m'a été réservé. Pendant cette année et demie passée au LORIA, j'ai découvert une équipe compétente, des collègues sympathiques, de véritables amis. Je remercie à ce titre Vincent Thomas et Alain Dutech pour leur sens de l'hospitalité. Je salue également tous les Maiamons et leurs amis, pour tous les bons moments passés à Nancy.

J'adresse un message d'amitié à toutes les personnes sans le concours desquelles ma thèse ne serait pas ce qu'elle est aujourd'hui. Je remercie Gilles pour ses idées, Matthieu, Hugo, Bruno(s), Gaëlle, Maroua, Aurélie et les membres l'équipe MAD pour leur aide pendant la dernière ligne droite. Je veux également saluer les collègues qui m'ont permis de faire un premier pas dans le monde de l'enseignement : Monique Grandbastien, Serge Stinckwich, Patrice Enjalbert. Je veux souligner l'efficacité de l'ensemble du personnel administratif du GREYC et du LORIA. J'ai une pensée toute particulière pour chacun de mes amis du labo à Caen.

Je témoigne toute ma reconnaissance à mes parents pour leur dévouement, leur soutien et leur amour. Merci papa d'avoir relu et corrigé mon mémoire. Je félicite Elsa pour la patience dont elle a fait preuve pendant cette unique et longue expérience qu'est la rédaction de thèse.

Table des matières

Table des figures

Introduction générale

L'exploration continue...

En 1492, Christophe Colomb découvre l'amérique ; en 1522 Magellan fait le premier tour du monde. Le spationaute Youri Gagarine entre le premier dans l'espace 1961, et en 1969, l'astronaute Neil Armstrong fait son premier pas sur la Lune. Chacune de ces découvertes a été rendue possible grâce à l'utilisation de nouvelles technologies : boussole, fusée... Les recoins encore inexplorés de l'univers sont nombreux, mais de plus en plus difficiles d'accès.

Aujourd'hui, l'homme fait appel à des entités de substitution afin de poursuivre cette conquête de l'espace. Des systèmes embarqués effectuent les missions là où nous ne pouvons plus aller. Depuis peu, la recherche dans le domaine de la robotique a permis de construire des entités mécaniques capables de se déplacer dans les endroits les plus inaccessibles : certains submersibles autonomes réussissent à plonger dans les fosses marines les plus profondes, des robots mobiles autonomes sont déployés la planète Mars pour y prendre des clichés et analyser ce sol inconnu. Des sondes et des satellites sont déployés pour explorer des planètes de plus en plus lointaines, voire en découvrir de nouvelles. Il existe également des projets de mise œuvre de robots capables de se déplacer dans de petites canalisations pour des missions de surveillance ou de nettoyage. On imagine également des robots capables de déminer un terrain, ou encore d'intervenir pour repérer puis sauver des blessés après un séisme.

Les systèmes embarqués doivent être autonomes, puisque une fois lancés, personne ne peut y avoir physiquement accès. Ils doivent donc être capables de se contrôler eux-mêmes. On parle alors d'agent autonome. L'agent, c'est celui qui agit, celui qui choisit la décision à prendre en fonction de la situation actuelle, de ses contraintes et de ses objectifs.

Dans les dernières décennies, des efforts considérables ont été menés pour concevoir de tels systèmes de contrôle, et à défaut de pouvoir créer un agent humainement intelligent, on préfère parler d'intelligence artificielle.

En inscrivant notre thèse dans le cadre de l'intelligence artificielle, nous avons l'ambition de présenter un tel mécanisme de contrôle flexible permettant à un système embarqué autonome d'effectuer certaines missions spécifiques. Le robot agira rationellement : il se comportera de façon à maximiser son intérêt tout au long de la mission. Cette mission sera un ensemble de tâches complexes, chacune ayant un intérêt particulier.

Les systèmes embarqués possèdent une quantité limitée de ressources, et la gestion de ces ressources est primordiale. Identiquement à un plongeur sans oxygène, un robot sans énergie est perdu. De plus, on ne peut jamais savoir avec exactitude combien de temps ni combien d'énergie un robot peut dépenser pour effectuer une partie de la mission. Une des principales contraintes de notre étude sera la gestion de la consommation de ressources sous incertitudes.

Parmi les nombreux systèmes de contrôle existants pour les agents rationnels, nous avons choisi de nous focaliser sur le raisonnement flexible, et plus particulièrement le raisonnement progressif. Nous lui portons un intérêt double : le modèle du raisonnement progressif permet,

grâce à son approche modulaire et hiérarchique, de décrire des modèles de tâches complexes en tenant compte des incertitudes de consommation de ressources. De plus, le comportement qui sera généré à partir de ce modèle est adaptatif : le robot pourra accomplir les tâches suivant plusieurs modalités, il pourra même interrompre une tâche pendant son exécution.

Notre apport dans ce contexte est double : le premier apport a consisté à étendre le modèle du raisonnement progressif, initialement prévu pour ne tenir compte que d'une seule ressource, à la gestion de plusieurs ressources consommables.

Nous proposons dans le deuxième apport un système de contrôle doublement adaptatif, permettant au robot de tenir compte des ressources consommables, mais aussi d'un changement éventuel de la séquence de tâches à effectuer pendant la mission, provoqué par exemple par un éboulement qui bouche le passage initialement prévu ou une défaillance mécanique interdisant l'utilisation d'un des outils embarqués.

Première partie

Techniques utilisées pour la planification sous incertitude et sous contraintes de ressources limitées

Chapitre 1

Présentation générale du problème

Introduction

Nous faisons appel aujourd'hui à des entités de substitution pour explorer certains endroits inaccessibles. Il faut doter ces systèmes embarqués d'un mécanisme de contrôle qui garantit leur autonomie.

Ce premier chapitre a pour vocation d'introduire les concepts généraux autour desquels s'organise notre thèse. Nous commencerons par introduire le concept d'Intelligence Artificielle, et la notion d'agent rationnel à laquelle nous nous rattachons. Nous présenterons ensuite certaines missions qui ont été effectuées par des robots autonomes sur Mars. Lors de ces missions, il y a des objectifs à atteindre, mais aussi des contraintes à respecter. Notre ambition est de fournir un mécanisme de contrôle permettant de gérer les ressources consommables d'un robot autonome qui aura comme objectif d'exécuter un ensemble de tâches complexes. Nous allons donc détailler les contraintes relatives à la consommation de ressources pour les systèmes embarqués, et en retenir un certain nombre. Nous motiverons également les approches que nous avons retenues pour l'élaboration d'un système de contrôle de robot autonome : planification sous incertitudes, raisonnement flexible et progressif. Nous finirons par proposer deux apports dans ce domaine : la gestion de ressources consommables multiples, et un système de contrôle qui s'adapte aux changements imprévus pendant la mission.

1.1 Intelligence Artificielle

La conception d'un robot intelligent autonome est directement liée au domaine de l'intelligence artificielle (ou I.A.). Ce domaine de recherche apparaît dans les années 50, suite à la création des premiers ordinateurs quelques années plus tôt. La définition du concept d'intelligence artificielle à longtemps fait débat dans la communauté. Les auteurs de [Russel et Norvig, 2003] catégorisent les différents courants de pensée selon deux questions fondamentales :

- Faut-il prendre en compte la manière de *penser* (la façon dont le système va atteindre ses objectifs) ou d'*agir* (la façon dont le système se comporte) ?
- Faut-il essayer de concevoir un système qui *reproduit* le comportement humain, ou de concevoir un système *rationnel* ?

Ces deux questions fondamentales divisent les définitions possibles de l'I.A. en quatre catégories, répertoriées dans un tableau tiré de [Russel et Norvig, 2003] :

Les systèmes qui pensent comme des humains	Les systèmes qui pensent rationnellement
Les systèmes qui agissent comme des humains	Les systèmes qui agissent rationnellement
"Le nouvel et intéressant effort pour faire penser des ordinateurs ... des machines dotées d'esprits, au sens propre." [Haugeland, 1985]. "[L'automatisation des] activités que nous associons à la pensée humaine, activités comme la prise de décision, la résolution de problèmes, l'apprentissage..." [Bellman, 1978]	"L'étude des facultés mentales à travers l'utilisation de modèles computationnels." [Charniak et McDermott, 1985] "L'étude des calculs qui rendent possible perception, raisonnement et action." [Winston, 1992]
"L'art de créer des machines assurant des fonctions qui requièrent de l'intelligence quand elle sont assurées par l'homme." [Kurzweil, 1990] "L'étude de la manière de faire faire aux ordinateurs des choses pour lesquelles jusqu'à maintenant, les hommes sont meilleurs." [Rich et Knight, 1991]	"L'intelligence computationelle est l'étude de la confection d'agents intelligents." [Poole *et al.*, 1998] "l'I.A. ... traite d'artefacts au comportement intelligents." [Nilsson, 1998]

Nous nous rattachons dans cette thèse au concept de **rationalité**. En sciences sociales (psychologie, psychologie sociale, économie), la rationalité caractérise une conduite cohérente, voire optimale, par rapport aux buts de l'individu. En informatique, un agent rationnel va agir en optimisant un certain critère de performance, en tenant compte des contraintes qui lui sont imposées pour atteindre ses objectifs, et ceci en fonction des connaissances et des perceptions qu'il a du monde. Cette définition s'éloigne certes beaucoup de la notion d'intelligence "humaine", mais a le mérite d'être intuitivement "mathématique". La difficulté pour le concepteur d'un agent artificiellement intelligent réside dans le choix du critère de performance et de la représentation des perceptions et des connaissances de celui-ci. En effet, ce sera toujours à partir de cette base mathématique que sera calculé un comportement rationnel. Mais seul un être humain sera, à notre avis, capable de juger si une intelligence émane de ce comportement rationnel ou non.

1.2 Le problème de la robotique autonome sur Mars

Durant notre thèse, nous nous sommes attachés au domaine de la robotique autonome, et nous l'illustrons avec le problème du robot explorateur martien qui a motivé l'ensemble de nos recherches. Depuis plusieurs dizaines d'années, la NASA projette d'envoyer un robot autonome sur la planète Mars, afin que celui-ci puisse renvoyer sur Terre des analyses (photographie, spectromètrie...) que l'homme ne peut pas faire lui-même. En 1996, ils lancent la mission PATHFINDER, lors de laquelle le robot SOJOURNER fera diverses analyses du sol martien. Le robot a évolué 83 jours, et a renvoyé 550 photos sur Terre avant de disparaître au mois de mars 1998.

Depuis, la NASA a lancé plusieurs missions similaires connaissant un succès plus grand. Les deux robots Spirit et Opportunity lancés lors de la mission Mars Exploration Rover (M.E.R.) durant le début de l'été 2003 continuent à explorer la planète Mars et renvoient régulièrement des photos et différentes analyses du sol vers la planète Terre

Cependant, nous avons continué à nous intéresser aux problèmes liés à la mission PATHFINDER, antérieure au début de notre thèse. Le rapport de la NASA disponible sur le site `http://www.nasa.gov` nous apprend que le temps de communication entre le rover et la planète Terre est d'environ 20 minutes, impliquant de la part du robot une autonomie totale au niveau de son contrôle. Le robot ne peut pas s'éloigner de plus de 500m de la station d'atterrissage, et dispose de batteries rechargeables par panneaux solaires, de divers instruments de mesure, et d'un détecteur d'obstacles.

Au départ, les missions qui incombaient au robot SOJOURNER se déroulaient dans un rayon de dix mètres autour du site d'atterrissage, car celles ci étaient dictées par des scientifiques qui depuis la planète Terre n'avaient accès qu'à ce que voyait la caméra du module d'atterrissage[1].

Le robot a perdu beaucoup de temps sur la planète Mars (environ 70% de son temps). Quand il était bloqué (par un rocher par exemple), il aurait fallu trouver une mission alternative pour éviter de lui faire perdre du temps. De plus, la consommation de ressources pour effectuer une tâche est souvent incertaine. Dans notre thèse, nous avons pour ambition de mettre au point un système de contrôle qui s'adapte à la fois à un changement de mission et à la consommation incertaine de ressources.

1.3 Une proposition de contrôle

La NASA et le JPL ont organisé plusieurs Workshop après l'échec de la mission PATHFINDER pour trouver un système permettant au robot de planifier et d'organiser ses actions sur la planète Mars de façon autonome. Dans le *second Workshop on Planning and Scheduling for Space*, Les auteurs de [Mouaddib et Zilberstein, 2000] ont proposé une approche permettant de

[1] qui ne pouvait distinguer que 10m aux alentours.

répondre partiellement aux problèmes posés par l'exploration de Mars avec un robot autonome. Celle-ci permet de :

- fournir une structure de tâches flexibles permettant au robot de trouver des alternatives pour mieux gérer ses ressources.
- tenir compte de la dépendance de consommation de ressources entre différentes tâches (car les ressources consommées pour une tâche ne seront plus disponibles plus tard).
- choisir la meilleure action possible, en tenant compte à la fois des ressources disponibles, du résultat déjà obtenu et des tâches restantes dans la mission (l'agent est rationnel)

Cette approche de contrôle pour un robot autonome est basée sur le raisonnement flexible et plus particulièrement le raisonnement progressif [Mouaddib, 1993], que nous présenterons plus tard.

Cependant, les auteurs proposent plusieurs problèmes ouverts :

- Comment gérer plusieurs ressources consommables ?
- Comment adapter rapidement la consommation de ressources en cas de changement imprévu[2] de la mission ?

Nous proposons des solutions à ces problèmes dans notre thèse.

Nous ne voulons pas limiter nos recherches au seul robot martien. Nous avons l'intention de fournir un système de contrôle pour un robot autonome pour que celui-ci puisse gérer les ressources consommables tout en maximisant la qualité du résultat produit par les différentes tâches qu'il effectuera lors de sa mission.

1.4 Problématique

Compte tenu des problèmes énoncés ci dessus, nous avons décidé de nous attacher à la classe de problèmes suivante :

Définition 1 *Un robot* ***autonome*** *évoluant dans un* ***environnement réel*** *et* ***dynamique*** *disposant de* ***ressources limitées****, doit* ***choisir*** *à* ***chaque instant*** *parmi un certain nombre de* ***tâches*** *celles qu'il va* ***effectuer*** *de façon à* ***satisfaire*** *au mieux la personne qui l'aura engagé.*

La plupart des termes utilisés dans cette définition, bien qu'ils soient naturellement compréhensibles, doivent être précisés ; nous allons le faire dans ce chapitre. Nous allons donc définir ce qu'est l'**autonomie**, voir les problèmes que posent les **ressources limitées** et finalement, les contraintes liées à un **environnement réel** et **dynamique**. Nous ne donnerons volontairement pas de formules mathématiques dans ce chapitre, afin de familiariser le lecteur avec ces notions. Les chapitres suivants nous donneront l'occasion de les développer en détail.

[2] Si un obstacle vient barrer le chemin initial de la mission par exemple.

1.5 Contraintes liées à notre problème

Le contexte dans lequel s'inscrit cette problématique apporte plusieurs contraintes fortes. Le robot va évoluer seul dans un environnement inaccessible à l'homme. La communication entre le robot et la ou les personnes qui lui proposeront une mission sera rare et coûteuse. Ceci impose d'avoir un robot autonome. Le robot, isolé, disposera d'une quantité de ressources limitée qu'il faudra gérer intelligemment pour mener à bien la mission. Enfin, le milieu réel dans lequel le robot évoluera engendre d'autres complications, que nous allons énumérer.

1.5.1 Autonomie

Notre objectif premier est de pouvoir confier une mission à une entité robotique sans que l'homme ait besoin d'intervenir sur place pendant la mission. Nous exigeons donc d'avoir un robot autonome. Un robot autonome est capable d'évoluer seul dans son environnement, sans aide extérieure. On lui fournit une mission et on le dote d'un mécanisme qui va lui permettre de mener à bien cette mission. Le besoin d'autonomie est illustré par l'exemple du rover martien. Un robot autonome doit donc être capable de se contrôler lui-même. Le contrôle aura un rapport avec la prise de décision et les ressources disponibles dans cette thèse.

1.5.2 Ressources limitées

L'autonomie des systèmes embarqués implique une limitation des moyens disponibles à bord. Une fois envoyé sur le lieu de la mission, on ne peut plus lui rajouter d'éléments physiques (comme une caméra, un bras, des roues). La durée de sa mission va être également limitée par les ressources consommables dont il dispose. Le mot *"ressource"* peut avoir plusieurs significations :

- il peut désigner un objet physique dont dispose le robot (par exemple une caméra) qui peut être disponible ou occupé.
- il peut désigner une quantité mesurable, et potentiellement consommable, comme l'énergie dans les batteries, la capacité mémoire disponible sur son ordinateur de bord.

Dans cette thèse, nous nous rattacherons uniquement à la deuxième signification, les *"ressources consommables"*. Un robot autonome ne dispose pas de ressources illimitées : même si les véhicules d'exploration planétaire ont la possibilité de recharger leurs batteries grâce à des panneaux solaires intégrés, un autre type de robots, les véhicules sous-marins autonomes [Turner, 2005] ne peuvent pas se pourvoir de tels avantages. La capacité de la carte mémoire embarquée sur le rover martien est elle-aussi limitée. Nous considérerons également dans notre contexte que le temps est une ressource consommable. Si le robot doit transmettre ses résultats avant une date butoir fixée, le temps écoulé pendant la réalisation d'une opération peut être considéré comme une consommation de temps.

Une mauvaise gestion des ressources peut s'avérer catastrophique : le robot sans batteries est perdu. Passer son temps à ne rien faire sur des missions d'exploration planétaire coûte de l'argent à ceux qui envoient le robot.

1.5.3 Contraintes liées à un environnement réel

Les contraintes liées à la robotique en environnement réel sont très nombreuses. Un robot ne peut pas raisonner sur l'ensemble des paramètres du monde réel, puisque ceux ci sont potentiellement infinis. Il doit de ce fait en avoir une représentation structurée, un modèle. Les problèmes de planification reposent la plupart du temps sur des représentations simplifiées du monde. Mais raisonner sur un modèle du monde, aussi fin et aussi précis soit-il, ne permet de garantir aucun résultat de manière certaine dans le réel. Les difficultés inhérentes au passage du modèle au réel sont liées au *temps* et à *l'espace*, mais aussi au *comportement* du robot.

La modélisation de l'espace

1. Les connaissances sur l'espace sont plus ou moins exactes. Le robot peut évoluer dans un environnement modélisé très précisément par ses contrôleurs, ou à l'inverse explorer un monde totalement inconnu.
2. La représentation de celui-ci peut être très précise ou au contraire floue. Ce flou peut être dû soit au manque de connaissances de la part des contrôleurs ou être le choix de simplifier le modèle. Cette abstraction évite de surcharger le robot en connaissances jugées inutiles.
3. La perception du monde par le robot lui-même, via divers capteurs, est souvent biaisée.

La dimension temporelle

1. Les temps de réflexion sont imposés : il faut prendre chaque décision au bon moment. On oppose dans la littérature des agents [Ferber, 1995] deux façons de décider : la **réaction** à la **délibération**. La délibération est une décision pour laquelle un laps de temps suffisamment long à été alloué au robot pour que celui-ci puisse choisir parmi les meilleurs choix possibles. La réaction intervient en cas d'urgence, face à certaines situations critiques. Par exemple, un robot qui se dirige vers un précipice doit s'arrêter dès qu'il s'en aperçoit ; il doit réagir. Nous nous plaçons dans un problème **temps réel** : le système doit délivrer une réponse dans des délais imposés.
2. Les évènements surgissent inopinément dans un problème **temps réel**. Le robot doit pouvoir s'adapter à cette contrainte forte en réagissant si besoin ou en délibérant si le temps le permet. Nous pensons néanmoins qu'il faut pouvoir assouplir ces deux notions de réaction et délibération, en s'accordant des délais de délibération échelonnés de façon à s'adapter

progressivement aux phénomènes de l'environnement. La dynamicité de l'environnement réel implique certaines contraintes fortes quant au temps alloué pour raisonner.

3. La durée des événements est elle aussi incertaine. Le robot peut avoir des connaissances sur celle-ci via la description des tâches qui lui sera fournie.

Le comportement du robot

Le mauvais déroulement d'une mission peut être dû au robot lui-même et non pas seulement à l'environnement dans lequel il évolue. Certaines défaillances mécaniques ou électroniques peuvent engendrer des dis-fonctionnements ou des pannes plus graves. Nous ne tenons pas compte des problèmes d'ordre mécanique ou électronique dans cette thèse.

Finalement, les contraintes retenues parmis toutes celles énoncées sont les suivantes :

- notre robot doit pouvoir décider seul, de manière rationnelle, être autonome,
- il doit savoir comment gérer la ou les ressources consommables dont il dispose, compte tenu de l'incertitude mesurée concernant cette consommation de ressources,
- il doit pouvoir s'adapter à un imprévu par rapport à la mission qui lui est initialement confiée.

1.6 Approches utilisées pour le contrôle de la mission

Ces différentes contraintes nous mènent à expliquer les concepts qui nous permettront de progresser vers une solution. Nous allons exposer ci-après selon quels critères notre agent prendra ses décisions pendant la mission. Ensuite nous proposerons un cadre permettant de raisonner en faisant face à l'incertitude liée à l'environnement réel. Nous présenterons le raisonnement flexible comme moyen efficace pour permettre au robot d'adapter son comportement pendant la mission, et finalement, nous exposerons les solutions que nous avons élaborées pour permettre au robot de gérer simultanément plusieurs ressources consommables, ainsi qu'une solution pour faire face à un changement éventuel de mission pendant son exécution par le robot.

1.6.1 Décision et rationalité

Le robot autonome que nous considérons est rationnel au sens de [Russel et Norvig, 2003],

Définition 2 *Agent rationnel :*

"For each possible percept sequence, a rational agent should select an action that is expected to maximise its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has."

Un agent rationnel doit donc à tout moment choisir l'action qui lui permet de maximiser son critère de gain. Puisque notre objectif est de faire effectuer au robot certaines tâches, nous nous donnons un moyen de mesurer le résultat des tâches ainsi effectuées. Nous attribuons à chaque tâche une qualité sous forme de donnée numérique avant que la mission ne commence. C'est sur cette base que le robot pourra calculer son critère de gain futur à maximiser. Mais ce critère dépend également des ressources qui seront consommées. Plus il consomme de ressources, moins le robot pourra effectuer de tâches par la suite. Il devra donc faire un compromis entre les qualités accumulées, et les ressources dépensées. La satisfaction des scientifiques qui enverront le robot en mission sera garantie par la rationalité.

1.6.2 Planification

Une fois qu'il connaît sa mission, le robot doit pouvoir décider quelle action effectuer. Une des façons d'y parvenir est de calculer un plan. Un plan est un ensemble d'actions qui le mènera vers son objectif en tenant compte des contraintes du problème. Pour planifier, le robot doit posséder un modèle du monde grâce auquel il pourra effectuer ses choix. Ce modèle doit notamment décrire les objectifs qu'il doit atteindre, ainsi que les contraintes auxquelles il devra faire face.

L'objectif de notre agent rationnel est de maximiser sur le long terme la somme des qualités qui seront produites par les tâches qu'il effectuera pendant la mission. Les contraintes sont liées aux ressources consommables embarquées dans le robot. Chaque tâche consomme plus ou moins de temps, d'énergie, ou de mémoire.

Dans le chapitre 2 , nous présentons la planification, dans un contexte général, puis dans un contexte probabiliste, puisque la consommation de ressources est sujette à l'incertitude.

1.6.3 Raisonnement sous incertitude

Autre contrainte à prendre en compte, l'incertitude. Cette incertitude concerne ici la consommation de ressources (temps, énergie...) disponibles pour mener la mission. Nous faisons l'hypothèse que cette incertitude sera mesurée avant la mission, par les créateurs du robot. Cette hypothèse est nécessaire pour rentrer dans un cadre de planification. Les processus décisionnels de Markov permettent de modéliser un environnement probabiliste. Ils sont utilisés depuis le début des années 90 dans le domaine de l'I.A. [Puterman, 1994] et nous les présenterons dans le chapitre 2. À partir d'un modèle sous forme de MDP et d'un critère à maximiser il est possible de calculer une (ou plusieurs) stratégie optimale. La programmation dynamique [Bellman,

1957] est un des moyens qui permet de trouver des algorithmes capables de fournir ces stratégies. Seulement, nous verrons que lorsque le problème posé devient de taille importante, les techniques permettant de calculer classiquement cette stratégie optimale ne suffisent plus. La capacité de calcul des machines actuelles[3] ne permet pas de trouver dans un temps correct la stratégie à adopter si la mission comporte un grand nombre de tâches. Pour ces raisons, il faut avoir recours à des méthodes de calcul spécifiques. Certaines de ces techniques consistent à produire des stratégies suboptimales mais cependant acceptables, via des calculs approchés. Nous présenterons une partie de ces méthodes dans le chapitre 3. Nous nous inspirons de certaines de ces techniques dans le chapitre 8.

1.6.4 Raisonnement flexible

Le raisonnement flexible dispose d'un mécanisme qui lui permet d'adapter dynamiquement la qualité de la solution aux ressources disponibles. [Mouaddib, 1993, Zilberstein, 1993, Boddy, 1991]

Il nous faut un système de contrôle interruptible, pour que le robot puisse abandonner une tâche s'il se rend compte qu'il perd du temps sur une tâche qui ne sera plus utile, ou impossible à achever. Nous voulons aussi une approche modulaire pour l'exécution des tâches, c'est-à-dire un système qui permette de réaliser les tâches de façon adaptative : passer plus de temps sur une tâche pour obtenir plus de qualité en utilisant des outils plus lents mais plus perfectionnés si le robot juge qu'il en a le temps (en fonction de son critère d'évaluation de gain), ou l'effectuer rapidement pour obtenir une qualité faible mais conserver des ressources pour la suite. Par exemple, quand le robot décide de faire une photographie, il peut d'une part, prendre le temps de positionner correctement son objectif, choisir un niveau de zoom, puis utiliser une résolution haute définition avant de stocker la photo prise sur son disque. D'autre part, s'il juge ne pas avoir le temps, il peut simplement viser son sujet sans s'arrêter, et prendre une photo floue, en basse résolution, et économiser de la place mémoire.

Le raisonnement progressif s'inscrit dans le raisonnement flexible, et répond à nos exigences, nous le présenterons dans la deuxième partie de cette thèse.

1.6.5 Notre apport

Suite aux attentes de [Mouaddib et Zilberstein, 2000], nous proposons une solution aux problèmes suivants (dans le cadre du raisonnement progressif) :

1. Étendre le problème du contrôle de la consommation d'une seule ressource à plusieurs ressources.

[3] et encore moins celle des systèmes embarqués.

2. Trouver un algorithme efficace (et temps réel) pour approcher la fonction de valeur qui permette de prendre rapidement une bonne décision en cas de changement soudain de mission.

De plus, nous avons conçu une expérience originale avec un robot *Koala*, pour valider concrètement l'utilisation du raisonnement progressif. Dans le scénario que nous proposons, le robot doit renverser des quilles en les poussant. L'objectif du robot, qui agit de manière rationnelle, est de renverser les quilles qui lui rapportent le plus de points dans un temps imparti. Cette expérience montre qu'il est possible, d'une part, de modéliser simplement une mission robotique à l'aide du raisonnement progressif et que d'autre part, le comportement obtenu est pleinement satisfaisant.

1.7 Plan de la thèse

Nous venons d'exposer dans ce chapitre le cadre dans lequel nous effectuons nos recherches : rationalité, robotique autonome, contrôle de consommation de ressources multiples, planification, raisonnement sous incertitude, et raisonnement flexible et progressif.

La suite de la **première partie** de ce mémoire est une présentation générale de la planification probabiliste et des processus décisionnels de Markov. Nous introduisons le problème de la planification dans le chapitre 2 sous forme d'historique avant de nous orienter vers le problème de la planification sous incertitude. Un modèle mathématique issus de la théorie économique, les processus décisionnels de Markov, est utilisé depuis le début des années 90 dans le cadre de la planification probabiliste. Nous y consacrons donc la dernière partie de ce chapitre. Le chapitre 3 est une étude sur la résolution de processus décisionnels de Markov de grande taille. Beaucoup de problèmes réels souffrent en effet de l'explosion combinatoire de l'espace d'états sur lequel est résolu le MDP. Nous présentons certaines techniques issues de la littérature comme la factorisation, l'agrégation et la décomposition de MDPs.

La **deuxième partie** de ce mémoire constitue le cœur du formalisme utilisé pendant cette thèse : le raisonnement progressif. Nous présenterons ses fondements, le raisonnement flexible, ainsi que plusieurs applications potentielles dans le chapitre 4. Nous formalisons le raisonnement progressif tel qu' A.I. Mouaddib et S. Zilberstein l'avaient proposé dans le chapitre 5. Nous donnons les algorithmes qui permettent de fournir un comportement rationnel à partir de ce modèle et nous mettons en avant le principe du coût occasionné qui permet de séparer le raisonnement en deux temps : contrôler les dépenses de ressources actuelles en fonction de la valeur qu'elles auraient pu fournir dans le futur. Nous proposons finalement une extension aux ressources multiples dans le chapitre 6. Nous montrons dans ce chapitre comment contourner le problème de la malédiction de la dimension dû à l'explosion combinatoire de l'espace d'états.

Nous consacrons la **troisième partie** de cette thèse au contrôle de consommation de res-

sources dans un environnement changeant. Nous présentons le problème du contrôle adaptatif de consommation de ressources dans un environnement changeant dans le chapitre 7. Dans cette partie, nous restreignons le raisonnement progressif à une seule ressource. Le chapitre suivant propose une façon de calculer rapidement une valeur approchée du gain espéré en fonction des ressources restantes pour contrôler localement la consommation de ressources. Cette approche est validée par une comparaison des différents comportements obtenus.

Finalement, la **dernière partie** sera consacrée à valider nos apports. Nous commençons par proposer un scénario pour illustrer le problème de contrôle via le raisonnement progressif : le robot bowling. Cette expérience a été menée avec un robot réel, le robot KOALA, acquisition de l'équipe MAD du GREYC. Les deux chapitres suivants valident expérimentalement les deux apports majeurs de notre thèse qui sont l'extension du raisonnement progressif à de multiples ressources et le contrôle de consommation de ressources dans un environnement dynamique.

Conclusion

Explorer une zone inconnue, inaccessible à l'homme pose, comme nous l'avons vu, de nombreux problèmes. Le déploiement de systèmes embarqués autonomes permet de poursuivre la découverte du monde qui nous entoure, mais ceux-ci ne peuvent pas être contrôlés directement par l'homme. Le domaine de l'Intelligence Artificielle offre des solutions intéressantes pour concevoir des robots autonomes capables de décider comment accomplir une mission qu'un humain aura auparavant modélisée. Énormément de problèmes sont directement liés à ce type de missions. Le robot doit agir et décider seul. Il doit pouvoir satisfaire le personnel pour qui il travaille en accomplissant au mieux sa mission. Ses ressources embarquées sont limitées, les ressources consommables doivent être intelligemment utilisées. Finalement, le robot doit se représenter au mieux l'environnement réel dans lequel il évolue, et tenir compte des aléas qui en découlent. Nous allons dans cette thèse exploiter le raisonnement progressif pour permettre à un robot de contrôler et d'adapter sa consommation de ressources afin d'accomplir une suite de tâches complexes.

Chapitre 2

Planification sous incertitude

2.1 Qu'est-ce que la planification ?

Planifier, c'est raisonner avant d'agir. C'est un processus de délibération explicite et abstrait qui choisit et organise les actions en anticipant leurs résultats escomptés.

Ceci est la première définition[4] intuitive de la planification de [Ghallab *et al.*, 2004]

Selon ces mêmes auteurs, dans la vie de « tous les jours » nous ne planifions pas toutes nos actions et nous ne sommes pas forcément conscients de la phase de délibération qui précède nos choix.

On réserve généralement la délibération à des tâches nouvelles, ou complexes, qui ont besoin d'être organisées. C'est le cas lorsque le problème à résoudre présente des contraintes, notamment si les actions présentent des risques en perte d'argent, de temps, ou de matériel. On n'essaye d'ailleurs pas forcément de trouver le plan qui nous donnera le résultat optimal, mais on se contentera [Simon, 1996] souvent de plans « réalisables ».

Dans notre cas, nous allons planifier pour faire résoudre par un agent un problème complexe dans un environnement, avec des **contraintes**, et des **objectifs** à atteindre. Le processus de délibération se base sur un **modèle** supposé **connu** du système[5] avant le raisonnement. On cherche ensuite à trouver un **comportement** pour l'agent permettant de résoudre ce problème (figure 2.1).

Ce chapitre est divisée en deux grandes parties : nous présentons un aperçu du domaine de la planification dite « déterministe », après quoi nous nous intéresserons au domaine de la planification probabiliste, pour finir par présenter le modèle sur lequel nous nous sommes basés pour résoudre notre problème de planification robotique, les processus décisionnels de Markov.

[4] Traduite de l'anglais.

[5] Le système est l'ensemble agent et environnement.

FIG. 2.1 – Planifier

2.2 Planification classique versus probabiliste

Il existe deux grands domaines de planification : la planification dans un monde déterministe dite **classique** et la planification **non déterministe**, qui prend en compte les incertitudes liées au monde dans lequel l'agent évolue. On situe généralement le début de la recherche en planification classique vers le début des années soixante avec le General Problem Solver (GPS) de Newell et Simon [Newell et Simon, 1963], [Newell et Simon, 1972] qui se fonde sur la résolution du problème sans réelle représentation des connaissances et le QA3 de [Green, 1969] qui lui se base sur une représentation formelle logique d'un problème de planification. STRIPS [Fikes et Nilsson, 1971] marque l'avènement de l'histoire moderne de la planification avec un langage de représentation des connaissances élaboré (description des opérateurs, des états). La représentation du monde des cubes reste l'exemple le plus connu (figure 2.2).

STRIPS est un planificateur d'ordre total, il suppose que pour atteindre un but, il faut spécifier et atteindre les sous-buts du problème, et que ceux-ci sont indépendants[6] et peuvent être résolus dans n'importe quel ordre, sans retour arrière sur les choix effectués. Or, pour certains problèmes très simples[7] comme celui de la figure 2.2, Sussman [Sussman, 1974] fait apparaître que certains sous-buts ne sont pas indépendants, et qu'il faut pouvoir revenir sur ses choix. C'est ainsi qu'apparaissent les planificateurs d'ordre partiel, comme NOAH [Sacerdoti, 1975], qui utilise des stratégies de moindre engagement[8]. Dans les années 80, Chapman [Chapman, 1987] introduit le planificateur TWEAK. Il étoffe le langage STRIPS en y ajoutant des contraintes aux actions sous forme de préconditions et de postconditions. TWEAK se base sur un critère de vérité qui permet de vérifier que le plan ne viole pas le système de contraintes susnommé. Une vaste étude a été menée sur le raffinement de cette technique, jusqu'à l'arrivée de la planification par compilation, qui se base sur une représentation d'un problème de planification en une structure particulière, et sur une résolution inspirée de la démonstration de théorème[9] et des prouveurs SAT (comme SATPLAN).

Dans les années 90 apparaît GRAPHPLAN [Blum et Furst, 1995], [Blum et Furst, 1997], qui transforme un problème de planification en une structure appelée graphe de planification.

[6] On appelle cela hypothèse de linéarité.
[7] Plus connu sous l'anomalie de Sussman.
[8] Il s'agit de retarder le plus possible le retour en arrière sur l'ordonnancement des actions.
[9] Comme l'avait fait Green avec QA3.

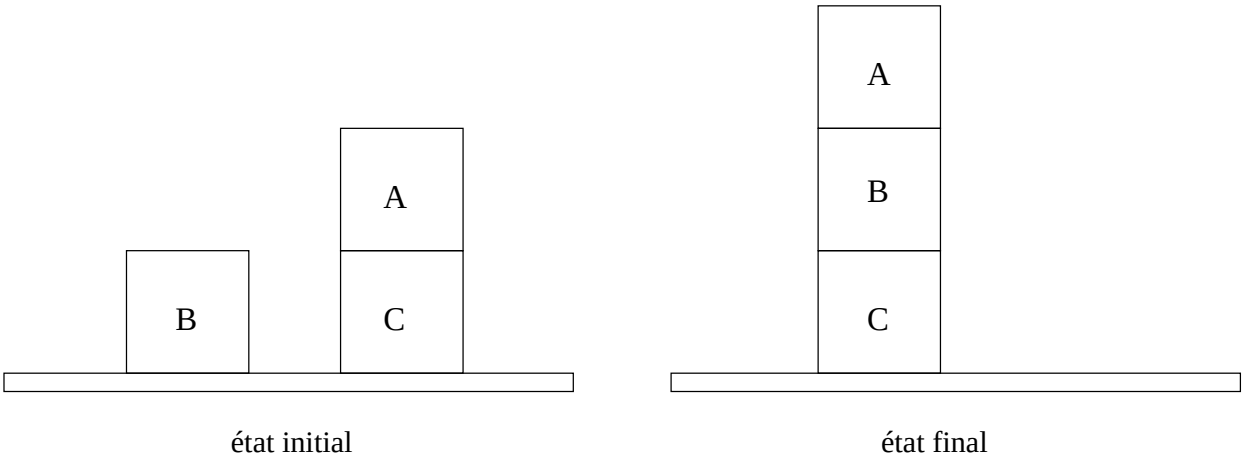


FIG. 2.2 – Un problème de planification dans le monde des cubes

Le problème des cubes. *Ce problème classique consiste à trouver un plan pour passer d'une situation de départ à une situation d'arrivée. On dispose pour cela d'une pince virtuelle qui peut déplacer un cube libre (sans cube au dessus) vers la table ou par dessus un autre cube. Dans notre exemple, il s'agit de passer de « A est sur C, B est sur la table, C est sur la table » à « A est sur B, B est sur C, C est sur la table ». Le problème se résout mal si on commence par satisfaire « A est sur B » en déplaçant A qui est libre sur B, car B devient non-libre, et on ne peut plus le déplacer sur C. Trouver l'ordre dans lequel les actions doivent être effectuées n'est donc pas trivial si on décide de ne jamais revenir sur ses choix.*

GRAPHPLAN est aujourd'hui un des tournant important dans le domaine de la planification classique. Nous conseillons la lecture du récent ouvrage [Ghallab *et al.*, 2004] pour plus de précisions sur la planification.

On peut résoudre un grand nombre de problèmes grâce à la planification classique, qui ne se limite évidemment pas au monde des cubes. Cependant, elle s'adapte assez mal au monde de la robotique et à tous les problèmes d'agents autonomes évoluant en milieu réel. Il existe en effet un degré d'incertitude lié aux actions de l'agent sur l'environnement. Cette incertitude est autant liée au résultat de l'action qu'à la manière dont celle-ci s'est déroulée. Le résultat d'une action entreprise peut par exemple être un échec complet ou une réussite totale. Cette action peut durer plus ou moins longtemps. Des extensions à la planification ont été effectuées pour prendre en compte l'incertitude comme PSO, une extension de STRIPS par [Hanks, 1993, Hanks et McDermott, 1994, Kushmerick *et al.*, 1995].

Notre problème, que nous détaillerons dans la deuxième partie de cette thèse, consiste à contrôler une consommation de ressources incertaine. Un processus décisionnel de Markov est un formalisme mathématique qui permet de modéliser un environnement soumis à des incertitudes. La communauté des chercheurs en intelligence artificielle s'y intéresse depuis le début des années

90 [Puterman, 1994]. Cependant, avant de présenter directement ce formalisme, nous allons introduire quelques concepts génériques de la planification pour un agent dans un environnement incertain.

2.3 Planification probabiliste

Nous allons introduire les concepts relatifs à la planification probabiliste [Littman, 1996]. De nombreux paramètres entrent en compte, et on peut faire ou ne pas faire certaines hypothèses sur le système qui modifient la façon dont le problème sera résolu. Nous allons donc énumérer pendant cette présentation les différentes hypothèses auxquelles nous nous tiendrons par la suite pour notre problème de planification.

2.3.1 Environnement

La planification à pour but d'élaborer un comportement pour un agent en supposant que l'on a un modèle du système.

État

Un **état** est un ensemble de descripteurs appartenant au modèle du système (environnement + agent). **Exemple** : dans une grille bidimensionnelle (figure 2.3), l'état peut prendre la forme (X, Y).

Observabilité

L'agent peut avoir une connaissance totale ou partielle de l'état dans lequel il se trouve : on parle alors d'observabilité ou de non observabilité. Nous faisons l'hypothèse pour toute la suite de cette thèse que **l'état est observable**[10].

Action

Nous faisons l'hypothèse que le temps est subdivisé en étapes d'actions. A chaque étape `t`, l'agent effectue une action, qui aura un effet sur l'environnement. En résultera un nouvel état que l'agent observera après l'issue de l'action (figure 2.4).

On peut limiter le nombre d'étapes d'actions à un nombre fini `T` que l'on appelle l'horizon. On dit alors que l'**horizon est fini**. Si le nombre d'étapes est infini, l'**horizon est infini**.

[10] Nous verrons dans le chapitre 5 que ce qui est observé est la quantité de ressources restantes, cette hypothèse forte est donc valable dans le cadre de cette thèse.

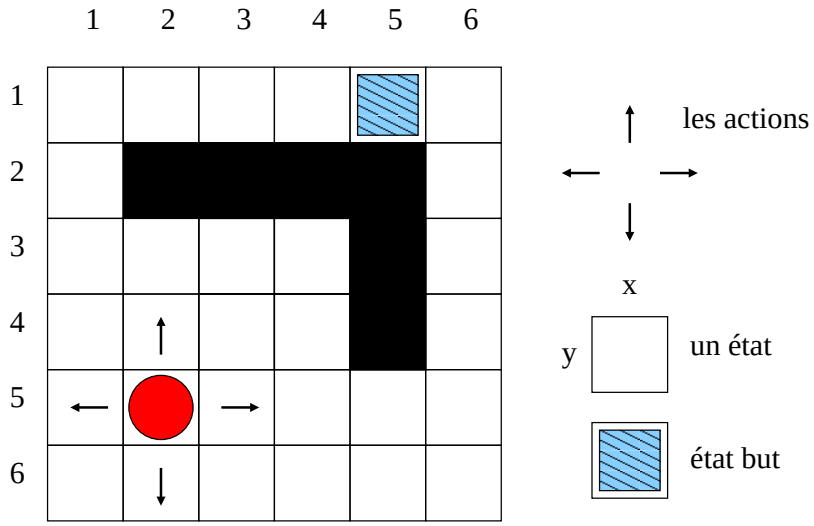


FIG. 2.3 – Un exemple : l'agent dans le labyrinthe

La figue 2.3 illustre un problème simple de planification : l'agent (le rond) doit trouver un comportement qui lui permette de rejoindre la case but hachurée. Il peut effectuer quatre actions représentées par les flèches. Son état actuel est d'être en (2,5).

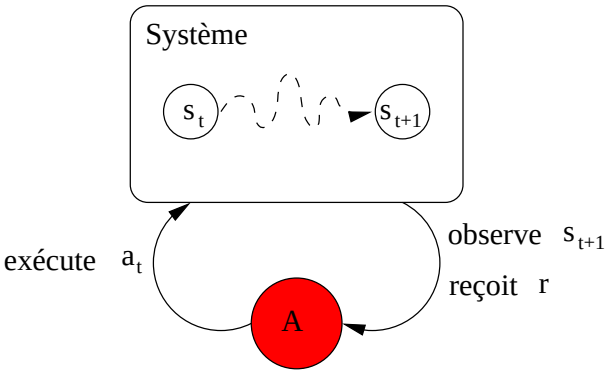


FIG. 2.4 – L'agent `A` agit puis observe l'état suivant

Incertitude

Il existe une incertitude sur l'issue des actions entreprises par l'agent. Cette incertitude fait partie du modèle du système en planification. On modélise donc cette incertitude au moyen d'une fonction de distribution de probabilité. Nous ferons l'hypothèse que cette fonction est constante dans le temps (on parle de modèles stationnaires).

La propriété d'observabilité reste tout de même valable : l'agent connaît l'état du système à un instant `t`, effectue une action à l'instant `t` sans connaître avec certitude (à cet instant) l'état

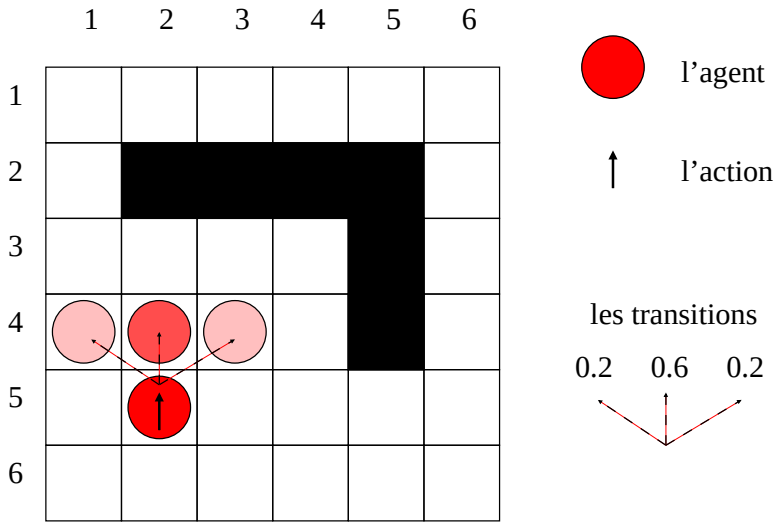


FIG. 2.5 – Modéliser l'incertitude

explications *pour la figure 2.5 : l'agent (le rond) exécute l'action ↑. Le résultat de cette action est incertain, il peut dévier légèrement vers la gauche ou vers la droite. La fonction de transition est connue sur ce schéma. En exécutant l'action ↑, l'agent a 6 chances sur 10 d'arriver sur la case du dessus.*

résultant. Cependant, à l'instant $\mathtt{t}+\mathtt{1}$, il observe et connaît le nouvel état.

2.3.2 Récompense

Il ne suffit pas de connaître le modèle du système pour élaborer un comportement pour l'agent. Afin de modéliser un objectif, on introduit une récompense. La récompense est une donnée numérique reçue à chaque étape de décision par l'agent. La récompense est une spécificité du problème à traiter par l'agent.

Par exemple, dans un problème où l'agent doit trouver la sortie d'un labyrinthe, la récompense sera nulle tant qu'il sera à l'intérieur, et 1 dès qu'il sera sorti (la récompense est associée à un état du système). Autre exemple : le robot aspirateur. On donne une récompense positive si l'agent effectue l'action aspirer dans l'état « poussière », une récompense nulle pour l'action « passer » dans l'état « propre » et une récompense négative sinon (la récompense est alors associée à la paire état action).

Il revient donc au concepteur de bien définir la fonction de récompense en fonction du problème à traiter.

2.3.3 Politique

La politique est une description du comportement de l'agent. C'est donc ce que nous allons chercher à optimiser. La politique indique l'action, ou une des actions à entreprendre par l'agent s'il est dans un certain état à une étape donnée. Pour certains problèmes, la décision ne dépend que de l'état de l'agent et plus de l'étape : on dit que la politique est **stationnaire**.

S'il n'existe pour chaque état qu'une seul action proposée par la politique, on dit que la politique est **déterministe**. Dans les autres cas elle est non-déterministe.

2.3.4 Critère d'évaluation

Après avoir décrit la façon de modéliser l'environnement, et la forme de notre solution, la politique, nous allons présenter les outils qui permettent d'évaluer cette solution. On se donne pour cela une **fonction objectif**, définie sur les états, les séquences d'actions et leur probabilité et qui produit une **valeur**. En planification, on cherche à trouver la politique qui permet de maximiser cette fonction objectif.

Cette fonction objectif peut prendre diverses formes, en fonction du problème à résoudre, et il revient donc également au concepteur du problème de bien choisir cette fonction de façon à optimiser le bon critère.

La valeur de la fonction objectif sera dans notre cas une somme sur le long terme des récompenses obtenues à chaque étape par l'agent dans le système. Quand l'horizon est infini, pour éviter que la somme ne diverge, on pondère la somme des récompenses par un facteur de convergence[11] $\gamma \in]0, 1[$.

Nous avons maintenant décrit comment :

- modéliser l'environnement
- modéliser les objectifs
- modéliser un comportement

Les processus décisionnels de Markov [Puterman, 1994] sont un cadre formel pour de tels problèmes de planification. Nous nous attachons aux restrictions faites plus haut : états observables[12], nombres d'états et d'actions finis. Dans notre application (voir chapitre 4 et 5), l'horizon sera fini. Nous présentons cependant le formalisme des MDPs dans les deux cas.

2.4 Les processus décisionnels de Markov finis observables

Les processus décisionnels de Markov sont issus de la théorie économique, et ont été étudiés dans les années 50. Cependant, la communauté de la planification et de l'Intelligence Artificielle en

[11] *Discount factor* en anglais.

[12] Les processus décisionnels de Markov partiellement observables ou POMDPs sont un cadre formel où les états sont non nécessairement observables.

général commence à s'y intéresser dans les années 90, pour produire des comportements d'agents rationnels. Nous nous rattachons donc à ce cadre pour présenter ce formalisme pluridisciplinaire.

Formellement, un processus décisionnel de Markov se définit comme suit :

Définition 3 *un processus décisionnel de Markov : (MDP)*

est un tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}r, \mathcal{R})$ *où :*

- $\mathcal{S}$ *est un ensemble d'états (fini et fixé)*
- $\mathcal{A}$ *est un ensemble d'actions (fini et fixé)*
- $\mathcal{P}r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0,1]$ *qui associe au triplet* $(\mathbf{s}, \mathbf{a}, \mathbf{s}')$ *la probabilité de passer de l'état* $\mathbf{s}$ *à l'etat* $\mathbf{s}'$ *en appliquant l'action* $\mathbf{a}$.
- $\mathcal{R} : \mathcal{S} \rightarrow \mathbb{R}$ *est une fonction de récompense*[13] *qui associe à chaque état* $(\mathbf{s})$ *une valeur numérique.*

2.4.1 Hypothèses

L'hypothèse de Markov (indépendance de l'historique)

Pendant la phase d'exécution, un agent passera successivement par plusieurs états, en exécutant des actions. On appelle historique $\{\mathbf{s}_0, \mathbf{a}_0, \ldots, \mathbf{s}_{t-1}, \mathbf{a}_{t-1}, \mathbf{s}_t, \mathbf{a}_t\}$ cette suite d'états actions, depuis l'instant 0 jusqu'à l'instant t. L'hypothèse de Markov stipule que la probabilité de passer à un état $\mathbf{s}_{t+1}$ en exécutant une action $\mathbf{a}_t$ ne dépend que de l'état courant $\mathbf{s}_t$ et de $\mathbf{a}_t$, donc elle est indépendante de l'historique :

$$\mathcal{P}r(\mathbf{s}_{t+1} = \mathbf{s}' | \mathbf{s}_0, \mathbf{a}_0, \ldots, \mathbf{s}_{t-1}, \mathbf{a}_{t-1}, \mathbf{s}_t, \mathbf{a}_t) = \mathcal{P}r(\mathbf{s}_{t+1} = \mathbf{s}' | \mathbf{s}_t, \mathbf{a}_t) \tag{2.1}$$

Récompense

Nous supposons également que la récompense n'est pas dépendante de l'historique du système :

$$\mathcal{P}r(\mathcal{R}^t | \mathbf{s}_0, \mathbf{a}_0, \ldots, \mathbf{s}_{t-1}, \mathbf{a}_{t-1}, \mathbf{s}_t, \mathbf{a}_t) = \mathcal{P}r(\mathcal{R}^t | \mathbf{s}_t, \mathbf{a}_t) \tag{2.2}$$

Stationnarité de la fonction de transition

Nous supposons également que la fonction de transition est stationnaire

$$\forall t, t', \mathcal{P}r(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t) = \mathcal{P}r(\mathbf{s}_{t'+1} | \mathbf{s}_{t'}, \mathbf{a}_{t'}) \tag{2.3}$$

[13] Nous l'avons dit dans la section précédente, on peut définir la fonction récompense comme : $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ ou encore $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$.

2.4.2 Formalisation de la politique

Une politique associe à chaque état une[14] action. On se restreint dans notre cas à une politique déterministe et stationnaire que l'on formalise comme suit :

$$\pi \quad : \quad \mathcal{S} \rightarrow \mathcal{A} \tag{2.4}$$

Cela signifie qu'à chaque état $\mathbf{s} \in \mathcal{S}$ est associé une action de l'espace $\mathcal{A}$. Cette politique π sera générée pendant la phase de calcul du plan et permettra de déterminer quelle action effectuer pendant la phase d'exécution du plan.

Pendant la phase d'exécution, l'agent adopte la politique π en suivant[15] l'algorithme 1 :

Data : un agent A, une politique π, un horizon T
1 **pour** t $\in$ [0..T] **faire**
2 A observe($\mathbf{s}$);
3 $\mathbf{a} = \pi(\mathbf{s})$;
4 A exécute($\mathbf{a}$);
5 **finpour**

Algorithme 1 : suivre la politique déterministe π

2.4.3 Agir de façon optimale

L'agent doit agir de façon à maximiser une fonction objectif sur le long terme. Cette fonction objectif associée à un MDP donne ce que l'on appelle un **problème de décision markovien**. Une fois la fonction objectif fixée, on cherche à trouver la politique qui optimise cette fonction.

Le critère que nous présentons[16] est la maximisation à long terme de la somme des récompenses obtenue. La fonction de valeur associée à ce critère à une définition différente si le problème à résoudre est à horizon fini ou infini.

En horizon fini

En horizon fini (en notant T l'horizon), le critère est l'espérance mathématique d'obtenir une somme de récompenses $\mathcal{R}(\mathbf{s}_t)$ à chaque étape t, à partir de l'état $\mathbf{s}_0$ de départ.

Définition 4 *Critère de performance*

$$E = \mathbb{E}[\sum_{t=0}^{T} \mathcal{R}(\mathbf{s}_t)|\mathbf{s}_0]$$

[14] Ou plusieurs.
[15] Dans le cas d'un horizon infini, la boucle est infinie.
[16] Ce critère sera celui qui sera retenu pour notre application.

En horizon infini

En horizon infini, l'espérance $\mathbb{E}$ peut être infinie. Pour éviter cela, on introduit un facteur $\gamma \in [0, 1[$ afin de faire converger la somme des récompenses.

Définition 5 *Critère de performance*

$$E = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t)|s_0]$$

Ce critère de performance défini, on peut évaluer une politique en la reliant à sa fonction de valeur $V : \mathcal{S} \rightarrow \mathbb{R}$

En horizon fini

En horizon fini (en notant T l'horizon), si on suit la politique π, l'espérance mathématique d'obtenir une récompense $\mathcal{R}$ au bout de T étapes (aussi appelée gain espéré) est caractérisée par l'équation de la définition 6 :

Définition 6 *Fonction de valeur*

La fonction de valeur $V^\pi : \mathcal{S} \rightarrow \mathbb{R}$ est

$$\forall s_0 \in \mathcal{S}, V^\pi(s_0) = \mathbb{E}[\sum_{t=0}^{T} \mathcal{R}(s_t)|\pi, s_0] \tag{2.5}$$

En horizon infini

En horizon infini, le gain espéré $\mathbb{E}$ peut être infini. Pour éviter cela, on introduit généralement un facteur $\gamma \in [0, 1[$ afin de faire converger le critère de la définition 6.

Définition 7 *Fonction de valeur*

La fonction de valeur $V^\pi : \mathcal{S} \rightarrow \mathbb{R}$ est

$$\forall s_0 \in \mathcal{S}, V^\pi(s_0) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t)|\pi, s_0] \tag{2.6}$$

2.4.4 Politiques optimales

On peut alors chercher à résoudre le problème de décision markovien en cherchant une politique qui maximise la fonction de valeur. Il peut exister plusieurs politiques optimales pour un même critère. Il existe dans la littérature de nombreuses méthodes permettant de trouver une politique optimale pour un problème de décision markovien, nous en présentons deux dans la section suivante.

2.5 Algorithmes pour obtenir une politique optimale π^*

Les algorithmes de programmation dynamique [Bellman, 1957] permettent de trouver une politique optimale pour un MDP donné. Le principe est d'itérer et de modifier soit :

- la valeur sur tous les états avec l'algorithme **value iteration** [Bellman, 1957]
- la politique avec l'algorithme **policy iteration** [Howard, 1960]

jusqu'à les faire converger vers une politique[17] optimale π^*. Les algorithmes que nous allons détailler sont donnés pour un horizon infini, les modifications à apporter pour un horizon fini sont données en note de bas de page.

2.5.1 L'algorithme value iteration

On calcule la valeur d'un état, qui est une mesure du gain espéré futur en utilisant l'opérateur de Bellman [Bellman, 1957] qui est une somme des valeurs des états suivants possibles pondérée par la probabilité de s'y retrouver en suivant la meilleure action possible. La fonction de valeur optimale, qui caractérise une politique optimale π^* à horizon infini est la suivante :

Théorème 1 *Fonction de valeur optimale. Soit $\gamma \in [0,1[$, la fonction de valeur optimale V^* : $\mathcal{S} \rightarrow \mathbb{R}$ est l'unique solution de l'équation*

$$\forall \mathbf{s} \in \mathcal{S}, V^*(\mathbf{s}) \quad = \quad \mathcal{R}(\mathbf{s}) + \max_{\mathbf{a}} \left(\gamma \sum_{\mathbf{s}' \in \mathcal{S}} \mathcal{P}r(\mathbf{s}'|\mathbf{a},\mathbf{s}) \times V^*(\mathbf{s}') \right) \tag{2.7}$$

On crée une fonction de valeur définie récursivement $V_{\mathtt{t}}$(pas encore optimale) pour espérer la faire converger vers V^*.

$$\forall \mathbf{s} \in \mathcal{S}, V_{\mathtt{t}}(\mathbf{s}) \quad = \quad \mathcal{R}(\mathbf{s}) + \max_{\mathbf{a}} \left(\gamma \sum_{\mathbf{s}' \in \mathcal{S}} \mathcal{P}r(\mathbf{s}'|\mathbf{a},\mathbf{s}) \times V_{\mathtt{t}-1}(\mathbf{s}') \right) \tag{2.8}$$

On peut réécrire simplement l'équation 2.8 de la façon suivante :

$$\forall \mathbf{s} \in \mathcal{S}, V_{\mathtt{t}}(\mathbf{s}) \quad = \quad F(V_{\mathtt{t}-1}(\mathbf{s})) \tag{2.9}$$

Dans la mesure où $\gamma \in [0,1[$, on montre que F est une contraction, et qu'en faisant l'hypothèse de Markov, il existe une solution à cette équation au point fixe qui est une politique optimale déterministe. Puisque nous avons supposé que les fonctions $\mathcal{P}r$ et $\mathcal{R}$ sont stationnaires, la politique obtenue est elle aussi stationnaire, elle ne variera pas au cours du temps.

Le principe à horizon infini de l'algorithme **value iteration** (algorithme 2) est le suivant (voir figure 2.6) : on initialise la valeur de tous les états à une valeur quelconque (par exemple à 0). À cet instant, $\forall \mathbf{s} \in \mathcal{S}, V_0(\mathbf{s}) = 0$. Ensuite on calcule une nouvelle valeur V_1 sur l'ensemble $\mathcal{S}$

[17] **value iteration** converge vers une fonction de valeur optimale qui permet de déduire une politique optimale.

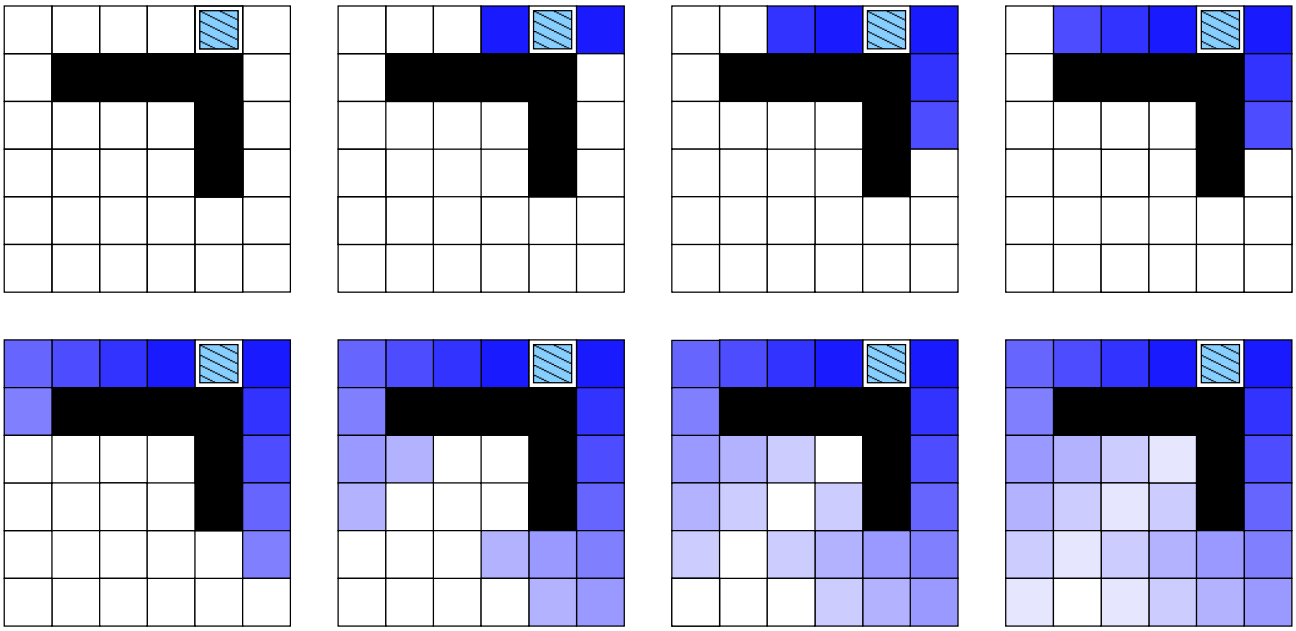


FIG. 2.6 – l'algorithme value iteration

explications *pour la figure 2.6 : Le principe de value itération est le suivant : on commence par initialiser les valeurs de tous les états à zéro (les états incolores). Ensuite, on va chercher pour chaque état la meilleure action à effectuer, puis mettre les valeurs des états à jour (la valeur de chaque état est représentée par un gris qui est plus sombre pour les plus grandes valeurs). Deux états mènent au but sur la grille de l'étape* 2. *On réitère le processus jusqu'à ce que les valeurs soient stables à un* ϵ *près. La politique optimale* π^* *est ensuite obtenue en affectant à chaque état l'action qui permettra de maximiser la fonction de valeur avec l'algorithme 3 page 31, consistant à faire une boucle supplémentaire sur l'ensemble des états.*

en appliquant l'opérateur de Bellman F de l'équation (2.9). Si la différence de valeur pour tous les états entre la fonction de valeur actuelle et la fonction de valeur précédente est inférieure à ϵ (la précision souhaitée), on renvoie cette fonction de valeur actuelle, sinon on réitère et on recalcule une autre fonction de valeur[18].

[18] En horizon fini T, il suffit de remplacer γ par 1 (pour le supprimer) et de s'arrêter au bout de T itérations, sans tenir compte de ϵ.

Données : un MDP $(\mathcal{S}, \mathcal{A}, \mathcal{P}r, \mathcal{R})$, $\gamma \in [0,1[$, $\epsilon \geq 0$
Résultat : $\widehat{V}$
1 Initialisation de V_0 à 0 sur $\mathcal{S}$
2 $\mathtt{t} = 1$
3 **répéter**
4 $\quad \forall \mathbf{s} \in \mathcal{S}, V_{\mathtt{t}}(\mathbf{s}) = \mathcal{R}(\mathbf{s}) + \max_{\mathbf{a} \in \mathcal{A}} \left(\gamma \sum_{\mathbf{s}' \in \mathcal{S}} \mathcal{P}r(\mathbf{s}'|\mathbf{a}, \mathbf{s}) \times V_{\mathtt{t}-1}(\mathbf{s}')\right)$
5 **jusqu'à** $\max_{\mathbf{s} \in \mathcal{S}}(|V_{\mathtt{t}}(\mathbf{s}) - V_{\mathtt{t}-1}(\mathbf{s})|) \leq \epsilon$;
6 **retourner** $\widehat{V} = V_{\mathtt{t}}$

Algorithme 2 : L'algorithme **value iteration** en horizon infini

Cet algorithme converge (voir la preuve dans [Puterman, 1994]) et la politique obtenue à partir de $\widehat{V}$ est optimale si ϵ est suffisamment petit. La complexité de cet algorithme est en $\mathcal{O}(|\mathcal{S}^2||\mathcal{A}|)$.

Données : un MDP $(\mathcal{S}, \mathcal{A}, \mathcal{P}r, \mathcal{R})$, $\widehat{V}$
Résultat : π^*
1 $\forall \mathbf{s} \in \mathcal{S}, \pi^*(\mathbf{s}) = \arg\max_{\mathbf{a} \in \mathcal{A}} \left(\gamma \sum_{\mathbf{s}' \in \mathcal{S}} \mathcal{P}r(\mathbf{s}'|\mathbf{a}, \mathbf{s}) \times \widehat{V}(\mathbf{s}')\right)$ **retourner** π^*

Algorithme 3 : Obtention de la **politique** avec la fonction de valeur.

2.5.2 L'algorithme policy iteration

La fonction de valeur V^π est une mesure du gain espéré si l'on suit une politique donnée π suivant. On mesure ce gain avec l'opérateur de Bellman. Avec un horizon infini[19] cette fonction de valeur prend la forme suivante :

Définition 8 *Fonction de valeur.*

$$\forall \mathbf{s} \in \mathcal{S}, V^\pi(\mathbf{s}) \quad = \quad \mathcal{R}(\mathbf{s}) + \gamma \sum_{\mathbf{s}' \in \mathcal{S}} \mathcal{P}r(\mathbf{s}'|\pi(\mathbf{s}), \mathbf{s}) \times V^\pi(\mathbf{s}') \tag{2.10}$$

[19] À horizon fini $\gamma = 1$ et $V_{\mathtt{T}} = F(V_{\mathtt{T}-1})$.

Données : un MDP $(\mathcal{S}, \mathcal{A}, \mathcal{P}r, \mathcal{R})$, $\gamma \in [0, 1[$, $\epsilon \geq 0$
Résultat : π
Initialisation quelconque de π
répéter
  $\pi' = \pi$
  pour $\mathbf{s} \in \mathcal{S}$ **faire**
    $V^{\pi}(\mathbf{s}) = \mathcal{R}\big(\mathbf{s}, \pi(\mathbf{s})\big) + \gamma \sum_{\mathbf{s}' \in \mathcal{S}} \mathcal{P}r(\mathbf{s}'|\pi(\mathbf{s}), \mathbf{s}) \times V^{\pi}(\mathbf{s}')$
  finpour
  pour $\mathbf{s} \in \mathcal{S}$ **faire**
    si $\exists \mathbf{a} \in \mathcal{A}, \mathcal{R}(\mathbf{s}, \mathbf{a}) + \gamma \sum_{\mathbf{s}' \in \mathcal{S}} \mathcal{P}r(\mathbf{s}'|\mathbf{a}, \mathbf{s}) \times V^{\pi}(\mathbf{s}') > V^{\pi}(\mathbf{s})$ **alors**
      $\pi'(\mathbf{s}) = \mathbf{a}$
    sinon
      $\pi'(\mathbf{s}) = \pi(\mathbf{s})$
    finsi
  finpour
jusqu'à $\pi' = \pi$;
retourner π

Algorithme 4 : L'algorithme **policy iteration** en horizon infini

Le principe de l'algorithme policy itération est de prendre une politique $\pi : \mathcal{S} \rightarrow \mathcal{A}$ que l'on va améliorer au fur et à mesure afin de converger vers une politique optimale. On commence donc avec une politique quelconque (on fait un assignement d'action arbitraire sur chaque état), puis on évalue chaque état de cette politique avec l'équation (2.10) de la définition 8.

Comme pour **value iteration**, cet algorithme converge (voir la preuve dans [Puterman, 1994]) et la politique obtenue est optimale. La complexité de cet algorithme est en $\mathcal{O}(|\mathcal{S}^2||\mathcal{A}|)$.

Conclusion

Planifier, c'est raisonner avant d'agir. Le début de ce chapitre présente les enjeux généraux de la planification : à partir d'un problème à résoudre, de contraintes, d'un modèle du monde, un agent délibère sur une stratégie à adopter pour résoudre ce problème. Viennent ensuite les étapes historiques relatives au domaine de la planification dite « *classique* ».

Notre problème de planification pour un robot autonome en environnement réel est soumis à certaines incertitudes, et nous avons choisi de modéliser le monde de ce problème avec le formalisme des processus décisionnels de Markov. Après avoir présenté la formalisation des MDPs, nous exposons le problème de décision markovien dont le but est, étant donné un critère à optimiser et une modélisation du monde sous forme de MDP, de trouver une politique, qui

associe à chaque état du monde une action à effectuer en fonction du critère à optimiser. Les MDPs que nous présentons sont finis, totalement observables et stationnaires. Nous présentons ensuite deux algorithmes classiques de la littérature des MDPs permettant d'obtenir une politique optimale pour un problème. La complexité des algorithmes classique est en $\mathcal{O}(|\mathcal{S}^2||\mathcal{A}|)$, où $\mathcal{S}$ est l'espace d'états possibles du monde et $\mathcal{A}$ l'espace des actions possibles dans ce monde.

Alors que la planification cherche un des meilleurs chemins faisables pour aller d'un état de départ vers un but, une politique permet de contrôler l'agent à chaque instant quel que soit l'état dans lequel il se trouve.

Le problème de décision markovien devient cependant difficile à résoudre pour des problèmes où l'espace d'états est grand. Les états sont décrits par de nombreux paramètres, qui par combinaison créent un espace de grande taille sur lequel le calcul de la politique devient long. Bellman avait souligné ce problème de la « *malédiction de la dimension* » ou « *curse of dimentionnality* » en anglais : le temps et l'espace mémoire nécessaires pour calculer une politique grandit exponentiellement avec, entre autre, le nombre de variables d'états.

Nous allons avoir des difficultés liées au calcul de la politique pour notre MDP dans notre problème de planification robotique, que nous aborderons essentiellement dans les chapitres 6 et 8. Aussi, nous allons présenter dans le chapitre suivant un panel de techniques dont nous nous sommes inspirés et qui permettent de résoudre les MDPs dits « de grande taille ».

Chapitre 3

Techniques de résolution pour les MDP de grande taille

Introduction

Nous avons vu précédemment que le principe général de la planification consiste à établir à l'avance une politique que l'agent devra suivre pour optimiser son gain espéré. Cependant, cette approche peut devenir "critique" :

- quand l'espace d'états ou d'actions est trop grand,
- quand le temps alloué pour calculer la politique est petit.

Le premier cas correspond à certains problèmes de planification où les états sont composés de beaucoup de variables différentes, sur des domaines détaillés, et aucune machine ne réussit alors à trouver la politique optimale en utilisant la programmation dynamique (dans un temps *correct*). On peut cependant avoir recours à la factorisation de MDP.

On rencontre le deuxième cas dans les problèmes temps réels, où il est nécessaire de trouver (ou de recalculer) rapidement une politique satisfaisante, et non plus optimale. En effet, certains types d'environnement dynamiques, où surviennent une multitude d'événements asynchrones plus ou moins prévisibles, demandent de pouvoir recalculer la politique à partir de l'instant où ceux-ci arrivent, et ce dans un temps plutôt court.

Le problème provient de la manière utilisée pour représenter le MDP à traiter. Les algorithmes de calcul de politique classiques sont en $\mathcal{O}(|\mathcal{S}^2||\mathcal{A}|)$ en programmation dynamique. Généralement, on représente la fonction de transition sous forme matricielle. Il y a pour chaque action une matrice carrée de l'ensemble $\mathcal{S}$ vers $\mathcal{S}$ à mettre en mémoire. Cela peut parfois devenir gênant si l'espace d'états est grand.

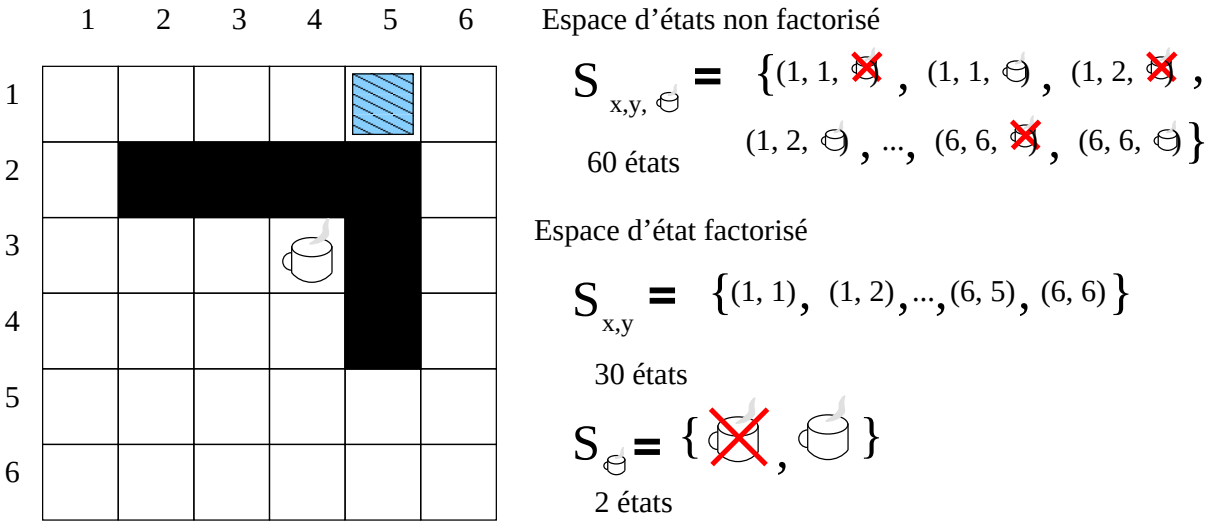


FIG. 3.1 – Factorisation d'un espace d'états

Explications *pour la figure 3.1 : dans ce nouveau labyrinthe, notre agent devra prendre un café à la machine à café, puis l'apporter jusqu'à la case finale (le carré hachuré). Il va donc y avoir six actions :* $\mathcal{A} = \{\uparrow, \downarrow, \leftarrow, \rightarrow, \textbf{prendre}, \textbf{poser}, \}$. *En conséquence les états du système sont la position et le fait de porter le café (ou pas). Si on les énumère tous, nous avons* $|\mathcal{S}| = |\Omega_{position}|.|\Omega_{caf}| = 30 * 2 = 60$ *états possibles. La représentation factorisée demande de manipuler seulement* $30 + 2 = 32$ *valeurs.*

3.1 Représentation structurée d'un MDP

3.1.1 Factorisation de l'espace d'états

La factorisation de l'espace d'états est une technique qui permet de représenter intelligemment la structure d'un MDP sans perte d'information. L'espace d'états n'est plus représenté de façon exhaustive mais comme un produit cartésien.

Prenons par exemple un état composé de N variables booléennes. La représentation exhaustive de $\mathcal{S}$ est de taille 2^N, et les matrices de transitions sont de taille 2^{2N}. Maintenant dans le cas général, une représentation factorisée de l'espace d'états où $\mathbf{s} = (X_1, ..., X_N)$ consiste à représenter séparément chaque variable X_i de l'état sur son domaine de valeurs possibles Ω_{X_i}, et de considérer que $\mathcal{S}$ n'est plus de la forme $\{\mathbf{s} = (X_1, ..., X_N)\}$ de taille $\prod_{i=1}^{N} |\Omega_{X_i}|$ c'est-à-dire un produit cartésien $\mathcal{S} = \Omega_{X_1} \times ... \times \Omega_{X_N}$, mais plusieurs espaces d'états Ω_{X_i} dont la taille totale est $\sum_{i=1}^{N} |\Omega_{X_i}|$. Un exemple simple de réduction de la structure est illustrée sur la figure 3.1 .

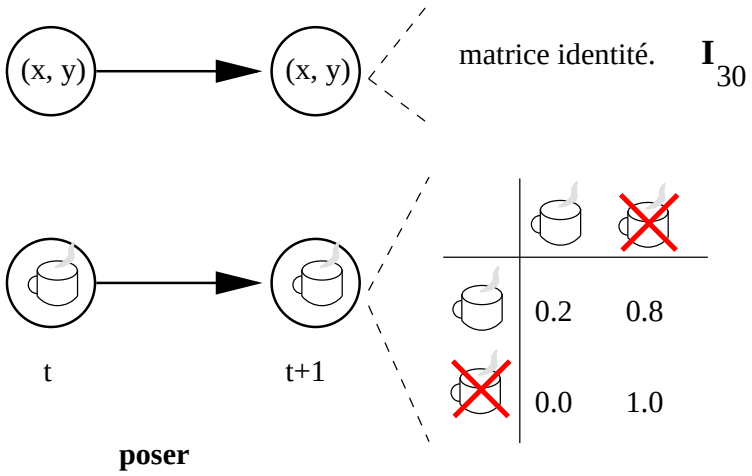


FIG. 3.2 – Factorisation de la fonction de transition : action **poser**.

Explications *pour la figure 3.2 : le fait de poser un café ne change pas la position de l'agent, donc on a une matrice de transition identité pour la position. Par contre pour la variable café, si l'agent avait un café à l'instant* $\mathtt{t}$*, il a* 8 *chances sur* 10 *de ne plus l'avoir à l'instant* $\mathtt{t}+\mathtt{1}$*, et* 2 *chances sur* 10 *de le garder en main (erreur de manipulation). S'il n'a pas de café avant de le poser, alors il n'aura toujours pas de café après. On a pour cette action une matrice de transition de taille* 30×30 *et une matrice de transition de taille* 2×2*. Sans réseau bayesien, on aurait eu une matrice de taille* 60×60*.*

3.1.2 Factorisation de la fonction de transition

La représentation des actions et des transitions change également. On utilise pour cela des réseaux bayesiens [Pearl, 1988]. Les réseaux bayesiens permettent une représentation compacte pour représenter une distribution de probabilités sous forme factorisée. Formellement, un réseau bayesien est un graphe acyclique dans lequel les nœuds sont des variables aléatoires et les arêtes orientées indiquent des probabilités de dépendance entre ces dernières. Pour représenter de manière factorisée la fonction de transition associée à une action donnée $\mathbf{a}$, on utilise des réseaux bayesiens temporels à deux niveaux, pour lesquels chaque nœud représente une variable du système à un instant $\mathtt{t}$ donné (le premier niveau) et les arêtes les probabilités de distribution vers une variable à l'instant $\mathtt{t}+\mathtt{1}$ (le deuxième niveau) après avoir effectué[20] $\mathbf{a}$. Au final, au lieu d'avoir pour chaque action une matrice de taille $\prod_{i=1}^{N} |\Omega_{X_i}|$ pour décrire les probabilités de transitions relatives à une action donnée, nous avons plusieurs matrices de taille $|\Omega_{X_i}|$, comme le montre les figures 3.2, 3.3 et 3.4 . Le papier de [Boutilier *et al.*, 2000] apporte beaucoup de précisions sur le sujet. les auteurs de [Kveton et Hauskrecht, 2006] s'intéresse également à la factorisation de MDPs.

[20] Équivalente à une dépendance temporelle quand $\mathbf{a}$ est choisie.

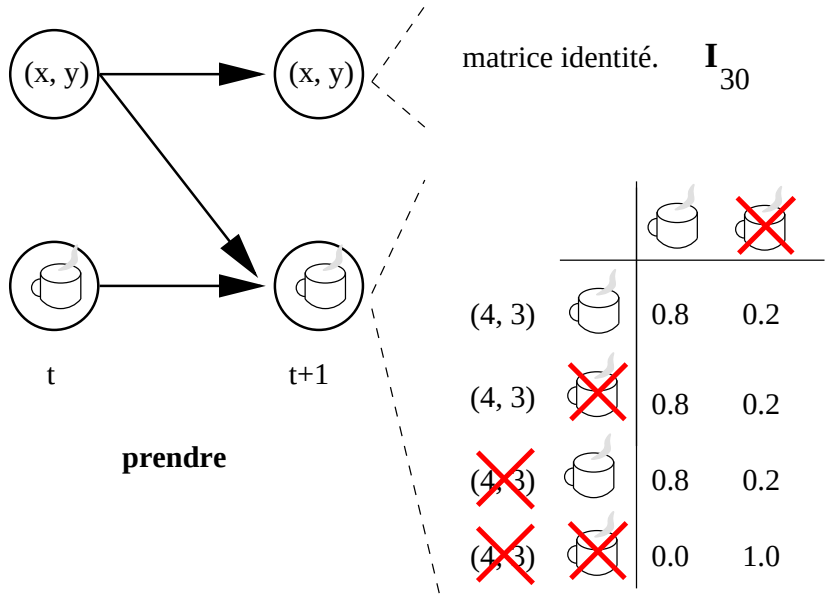


FIG. 3.3 – Factorisation de la fonction de transition : action **prendre**.

Explications *pour la figure 3.3 : le fait de prendre un café ne change pas la position de l'agent, donc on a une matrice de transition identité pour la position. Par contre la variable café à l'instant* `t + 1` *dépend de la position de l'agent (s'il est bien sur la case (4, 3)) et s'il avait ou pas un café à l'instant* `t`*. S'il n'est pas sur la bonne case, il n'a aucune chance d'avoir un café après avoir essayé de le prendre. Sinon, il a* 8 *chances sur* 10 *de réussir son action.*

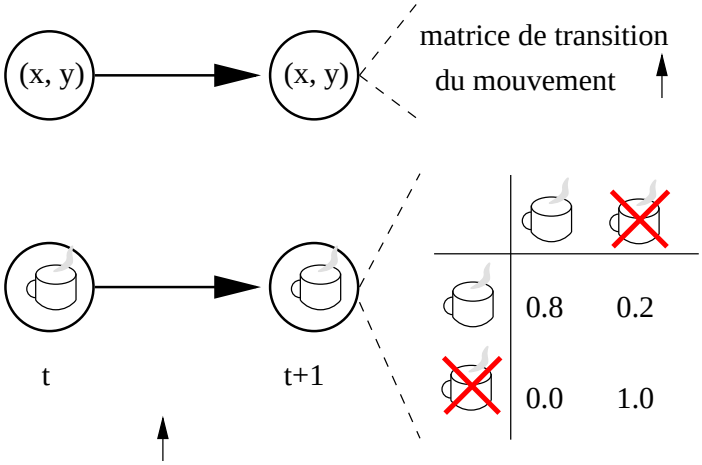


FIG. 3.4 – Factorisation de la fonction de transition : action ↑.

Explications *pour la figure 3.4 : le fait de bouger vers le haut entraîne, au niveau de la position, les mêmes conséquences si le robot a ou n'a pas de café. La matrice de transition pour la position est donc la même que pour une action ↑. Par contre, en bougeant, le robot peut*

renverser sa tasse. Si il avait du café avant de bouger, il a 2 *chances sur* 10 *de le renverser.*

3.2 Techniques de résolution approchée

Plusieurs techniques ont été développées au cours de ces dernières années pour outrepasser les inconvénients inhérents à la taille d'un MDP. On peut **abstraire** le raisonnement et la modélisation de celui-ci, c'est-à-dire le représenter de façon plus compacte, sans s'occuper des détails de chaque état, de chaque action. On peut également explorer le MDP avec une **heuristique** [Pearl, 1983], en ne s'occupant volontairement pas de certains *endroits* qui n'ont pas vraiment d'influence sur le résultat final. Enfin, on peut **décomposer** le MDP en petites parties qui seront résolues séparément, et que l'on recombinera finalement ensemble. Évidemment, on peut combiner plusieurs de ces astuces pour obtenir plus facilement le résultat escompté.

Toutes ces techniques offrent souvent un avantage et un inconvénient majeur : la politique pourra être calculée dans un temps raisonnable[21] , mais elle ne sera souvent qu'approchée. Pour savoir si une technique est avantageuse ou non, il faut pouvoir mesurer le compromis entre le temps que l'on gagne pour le calcul, et la qualité perdue pour la politique obtenue. Ceci n'est malheureusement pas toujours faisable. Nous allons présenter diverses techniques utilisées dans la littérature.

3.3 Abstraction

Abstraire un problème c'est trouver une représentation plus compacte, moins détaillée qui permet de raisonner plus rapidement sans rentrer dans les détails.

L'abstraction peut avoir lieu au niveau de l'espace d'états dans le cadre des MDPs. On appelle généralement ce type d'abstraction l'agrégation d'espace d'états.

Le principe d'agrégation est le suivant : au lieu de raisonner sur un ensemble d'états très grand, on regroupe ces états par paquets en macro-états, selon certains critères, généralement des variables d'états, et on recherche une politique sur ces macro-états, qui sont une nouvelle partition de l'espace original, tout en essayant d'altérer le moins possible la politique abstraite obtenue. La taille de l'espace des macro-états étant plus petit que l'espace d'états de départ, les algorithmes de recherche de macro-politiques donnent un résultat plus rapidement.

Les trois voies d'agrégation utilisées sont(voir figure 3.5) :

- L'abstraction **uniforme** qui s'oppose à l'abstraction **non-uniforme**
- L'abstraction **exacte** qui s'oppose à l'abstraction **approchée**
- L'abstraction **adaptative** qui s'oppose à l'abstraction **fixée**

[21] Qui correspond au temps au bout duquel nous souhaiterions obtenir un résultat ou une politique correcte.

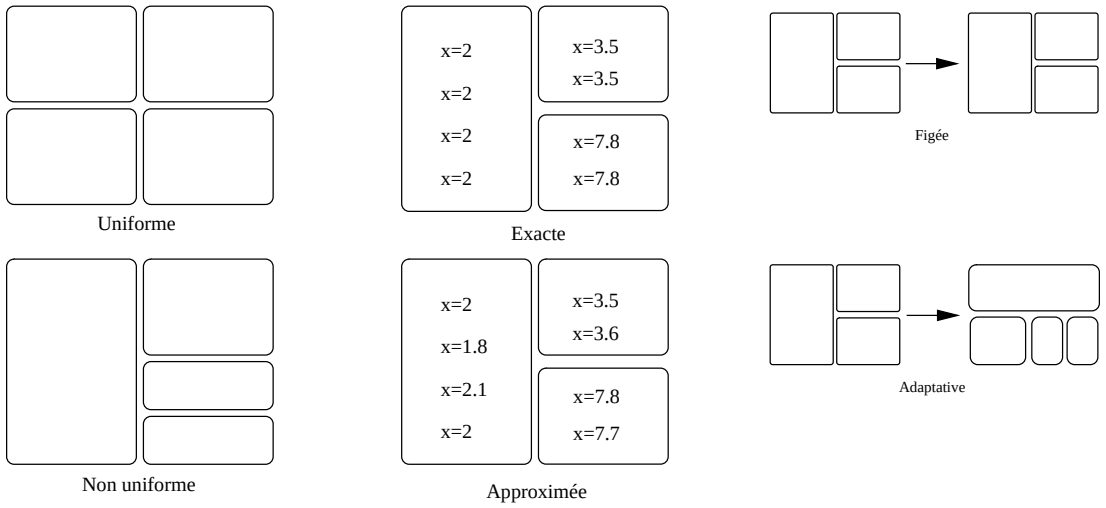


FIG. 3.5 – Les trois techniques d'agrégation de l'espace d'états

L'abstraction **uniforme** suppose que chaque macro-état représente la même quantité d'états de l'espace de départ.

Quand on regroupe des états vis à vis d'un certain critère, l'abstraction est **exacte** si tous les états ont la même valeur pour ce dit critère, on peut aussi regrouper des états pour lesquels cette valeur est proche, on dit alors qu'elle est **approchée**.

Si l'abstraction de l'espace d'états se fait en une seule fois, et ne change plus ensuite, elle est dite **fixée**. Si au contraire on peut réitérer le processus d'abstraction au cours de l'algorithme d'évaluation (de la politique, ou de la valeur des états) on dit que l'abstraction est **adaptative**.

3.3.1 Régression vers le but et planification classique

Si le problème comporte des états buts, on peut réutiliser les techniques de la planification classique pour essayer d'agréger l'espace d'état. En effet, dans certains problèmes, on peut regrouper certains états qui mèneront tous indifféremment l'agent vers son but. Le regroupement fonctionne de la façon suivante : on regroupe dans un premier temps les états qui représentent un but équivalent. Puis, par chaînage arrière, on recherche les états qui mènent à ce but (on les appelle les sous buts) et on les regroupe dans un macro-état sous but. Une fois encore, on cherche les états qui mènent au sous but, et ainsi de suite jusqu'à ce que l'un des macro-états finaux englobe les états de départs (ou tous les états si on cherche une politique exhaustive), ou que tous les états aient été regroupés et qu'aucun chemin valable n'ait été trouvé. Une telle représentation est donc plus compacte.

3.3.2 Plans abstraits

Pour simplifier un problème de planification donné, nous, êtres humains, sommes capables de ne pas prendre en compte un ensemble de paramètres "inutiles". Pour passer d'une pièce à une autre, nous ne cherchons pas à passer à travers les murs, ni même à rejoindre le fond de la pièce. C'est à partir de cette constatation que C. Boutilier et R. Dearden ont essayé dans [Dearden et Boutilier, 1997] de réduire un MDP en ne tenant pas compte des variables jugées inutiles. Leur travail se base sur l'idée de C.A. Knoblock[22] de classer les variables en fonction de leur importance pour le problème à résoudre. Ainsi, en modélisant a priori le problème sous sa forme STRIPS ou avec des réseaux bayesiens dynamiques, on peut supprimer non seulement certaines actions sans importance, mais également les récompenses associées. On aboutit à un plan abstrait, plus rapide à calculer que le vrai plan, mais aussi beaucoup moins précis. Les auteurs de ce travail montrent cependant qu'il est possible de se resservir de ce plan abstrait[23] pour démarrer un nouveau calcul de politique plus précis, ou comme heuristique pour calculer le plan.

3.3.3 Minimisation de modèles

On connaît pour la théorie des automates finis déterministes un moyen de retrouver l'unique automate fini déterministe minimal capable de lire un langage donné L. C'est en se basant sur cette idée de minimisation que T.Dean et R.Givan dans [Dean et Givan, 1997] réduisent leur MDP. Ils ne cherchent pas à obtenir le modèle minimal, mais seulement à tendre vers celui-ci. Les états sont agrégés par groupes d'états qui satisfont des propriétés communes. Cette agrégation se fait au fur et à mesure de manière adaptative.

3.4 Décomposition en série

L'idée de la décomposition provient du principe bien connu "*Diviser pour régner*". Plutôt que de calculer une politique sur l'espace des états $\mathcal{S}$, on compile des politiques par régions d'états plus petites que l'espace total initial. On les appelle politiques locales. Ensuite, on regroupe toutes les politiques locales de façon à recomposer une politique globale[24]. L'avantage de cette technique est le suivant : le MDP à décrire est divisé en MDPs locaux moins grands, donc le calcul de toutes les politiques locales est plus rapide que celui de la politique globale. La figure 3.6 illustre ces avantages. Il existe aussi des techniques de décomposition sur l'espace d'actions $\mathcal{A}$.

Reste maintenant à trouver comment effectuer chacune de ces étapes.

[22] Knoblock a travaillé sur l'abstraction pour des problème de type STRIPS déterministes.

[23] On peut retrouver la valeur de la politique optimale grâce à l'algorithme **value itération** en partant d'un espace d'états sous évalué.

[24] Cette politique globale ne sera bien entendu pas forcément optimale.

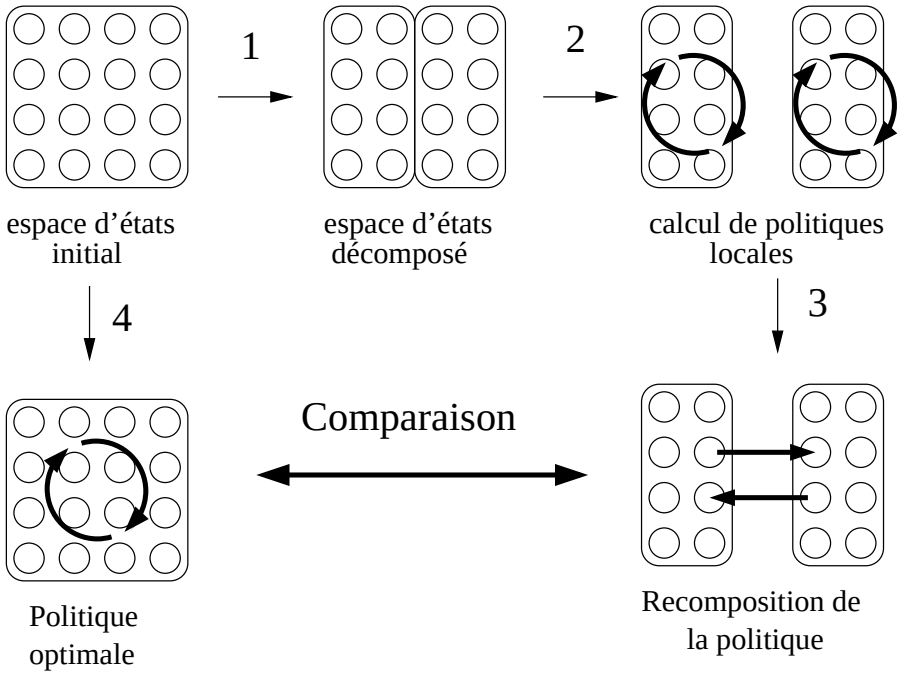


FIG. 3.6 – Illustration simplifiée de la décomposition d'un MDP

Explications *pour la figure 3.6 :*

1. *Partitionnement de l'espace d'états : il peut être fait à la main, ou automatiquement.*
2. *Calcul des politiques locales : si on considère les deux parties $\mathcal{S}_1$ et $\mathcal{S}_2$, la complexité de cet algorithme sera intuitivement en $\mathcal{O}((|\mathcal{S}_1|^2 + |\mathcal{S}_2|^2)|\mathcal{A}|)$.*
3. *Recomposition : il faut trouver un moyen de recomposer toutes les politiques locales en une politique globale qui tienne compte de toutes les régions.*
4. *Le calcul de la politique optimale n'est pas toujours faisable, c'est pour cela que l'on a recours à la décomposition. On peut néanmoins comparer la technique de décomposition avec la technique classique sur des problèmes de petite taille.*

3.4.1 Accessibilité du but

Dans certains problèmes de planification, l'état de départ, ou l'ensemble des états de départs est connu. Il y a, dans ce type de problème, des états qui ne seront pas visités pendant la phase d'exécution en suivant la politique obtenue classiquement. Il serait donc judicieux d'éviter le calcul de politique pour les régions qui ne seront pas visitées. La manière la plus simple pour détecter ces régions non accessibles est de lancer un algorithme de recherche en avant depuis les états de départs sur l'ensemble de l'espace d'états, et d'éliminer tous les états qui ne seront pas accessibles. À l'inverse, quand l'ensemble d'états buts est connu, on peut détecter

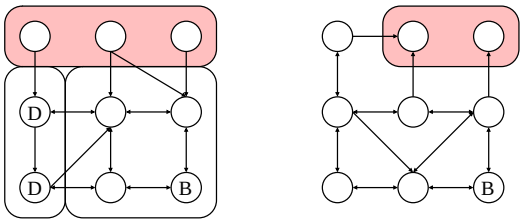


FIG. 3.7 – certaines régions inaccessibles dans un MDP

La figure 3.7 montre à gauche un problème où les états de départs sont connus (désignés par un D). Or, il est impossible d'aller dans la région du haut depuis ces états, inutile donc de la prendre en compte. Dans l'exemple de droite les buts sont connus (marqués par un B) on remarque que les états en haut à droite ne conduisent d'aucune manière au but, il ne devront donc pas être pris en compte pour la compilation de la politique.

certaines zones à partir desquelles il sera impossible d'aboutir au but. Pour trouver de telles zones, un algorithme de recherche en arrière est suffisant. On peut améliorer et combiner ces deux techniques, et s'inspirer de l'algorithme GRAPHPLAN de [Blum et Furst, 1995] comme l'ont fait les auteurs de [Boutilier *et al.*, 1998]. Sur ce principe, on peut répertorier certaines contraintes relatives aux états buts en transformant la représentation du MDP sous forme de réseaux bayesiens dynamiques. On repère de cette façon des états qui ne peuvent pas être atteints, tout simplement parce que certaines combinaisons de variables d'états ne figurent dans aucun ensemble de variable d'arrivée de ces réseaux.

Dans ce type de problème, la phase de décomposition (phase 1 de la figure 3.6) est la seule qui doive être effectuée : le seul MDP local pour lequel il faut calculer une politique (phase 2) est celui des états accessibles. Le reste n'est pas pris en compte, donc la recomposition (phase 3) n'a pas lieu.

On ne peut malheureusement pas toujours trouver de telles zones. Il existe cependant des techniques similaires qui exploitent la structure de certains MDPs pour les décomposer en sous parties.

3.4.2 Structures faiblement couplées

À défaut de ne jamais être visitées, certaines zones d'un MDP ne sont que faiblement liées. La figure 3.8 montre que pour passer de la région A à la région B, il faut passer par des états spécifiques. Les autres états de ces deux régions ne peuvent pas communiquer, c'est-à-dire qu'aucune action partant de ces états ne permet d'aboutir à un état de l'autre région (ceci est schématisé par un mur). On dit que ces zones sont faiblement couplées, ou faiblement communicantes.

La première étape consiste donc à repérer les parties faiblement couplées, et à partitionner

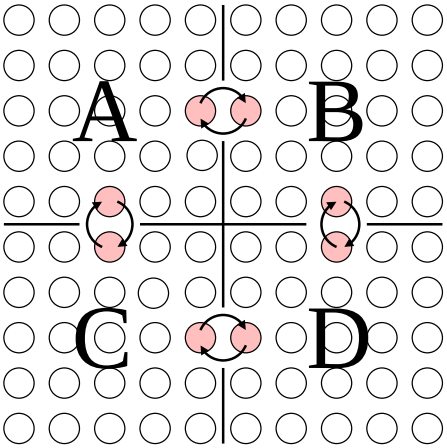


FIG. 3.8 – Certaines régions sont faiblement couplées dans un MDP

explications *pour la figure 3.8 : cette figure représente un espace physique de quatre salles séparées par des murs. On représente chaque position possible dans cet espace par un état (sous forme de rond). On peut déplacer le robot d'un état à un de ses états voisins uniquement. On distingue donc quatre régions faiblement couplées et huit états grisés par lesquels le robot doit passer pour transiter d'une région à une autre.*

l'espace d'état, comme le montre la figure 3.9. Ensuite il faut regrouper les états périphériques en un bloc appelé noyau. Ces états périphériques sont ceux qui permettent de communiquer avec les autres régions. La deuxième étape est la résolution de ce MDP décomposé : on commence par donner une valeur aux états du noyau, puis on compile une politique pour chaque MDP local, donnant ainsi de nouvelles valeurs aux états périphériques. On réitère ce procédé jusqu'à ce que les valeurs des états du noyau n'évoluent plus. La troisième étape est inclue dans la deuxième, la recomposition se fait au fur et à mesure de la mise à jour du noyau.

T. Dean et S. Lin montrent dans [Dean et Lin, 1995] comment résoudre ce type de MDP décomposé avec plus de détails. Trouver une décomposition automatiquement reste assez difficile. [Lin, 1997] a exploité la structure particulière de certains MDPs pour obtenir une décomposition automatique. R. Parr montre dans [Parr, 1998] qu'il est possible d'exploiter la structure faiblement couplée de certains MDPs (comme celui de la figure 3.9) en calculant une politique pour chaque region. On peut ainsi obtenir soit une politique recomposée optimale, soit une politque recomposée suboptimale, avec un écart théoriquement mesurable. R. Sabbadin propose dans [Sabbadin, 2002] une décomposition automatique de MDP basée sur des coupes minimales de graphes.

Pour ce type de décomposition, on peut aussi utiliser des macros actions qui sont généralement des politiques sur les régions. F. Teichteil détaille la décomposition en série dans son mémoire de thèse [Teichteil, 2004]. Il nous propose également une généralisation des techniques

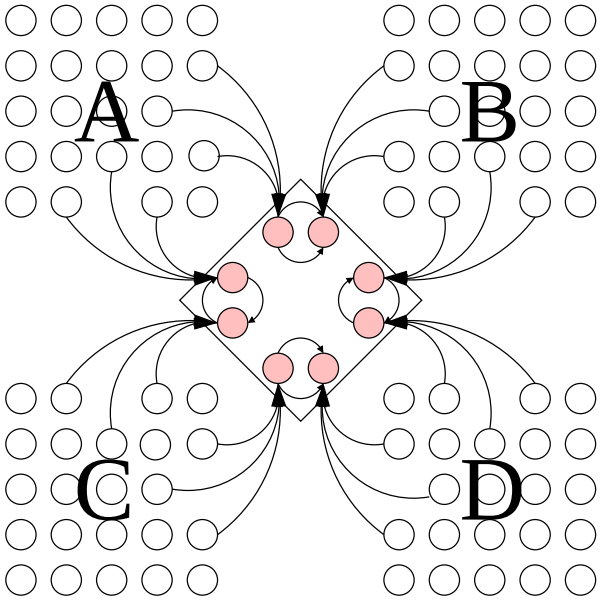


FIG. 3.9 – Noyau et zones décomposées.

explications *pour la figure 3.9 : on regroupe dans une même région que l'on appelle noyau l'ensemble des états périphériques (états grisés) qui permettent de transiter d'une région à l'autre. Les flèches matérialisent les transitions possibles entre deux états.*

de décomposition de MDPs énumérés aux MDPs factorisés, ainsi qu'un algorithme d'optimisation heuristique de MDPs factorisés ; tout ceci en utilisant des réseaux bayesiens dynamiques.

3.5 Approximation de fonction

On utilise les approximateurs de fonctions (en statistique) quand on veut apprendre/estimer une fonction : on a des échantillons, c'est-à-dire des couples (x_i, y_i) où x_i est un vecteur réel en P dimensions et y_i un réel ; on veut construire une fonction qui passe au mieux par ces points et qui donne une estimation de sa valeur pour tout autre point du domaine $\mathbb{R}^P$ (on y a recours lorsqu' on ne peut pas calculer analytiquement la fonction) ; on veut que l'erreur commise sur les points pour lesquels on doit prédire la fonction soit faible.

Dans le domaine de l'apprentissage par renforcement, domaine connexe à la planification sous incertitudes, on utilise un approximateur de fonction dès que l'espace d'états (ou l'espace état d'action) est trop grand (quelques millions) pour être représenté efficacement (en espace mémoire, mais surtout en terme de possibilité et de temps d'apprentissage) de manière tabulaire, voire continu. L'intérêt d'utiliser un approximateur de fonction est également d'avoir immédiatement une généralisation de l'apprentissage : avec une table, si on veut apprendre la valeur de la fonction pour une certaine entrée de la table, il faut explicitement obtenir cette valeur (ou utiliser des

mécanismes particuliers comme dans [Preux, 2004]) ; avec un approximateur de fonction, on apprend une estimation de la fonction pour tout point du domaine, même ceux par lesquels on n'est pas passé.

Pour approcher une fonction on peut utiliser des approximateurs de fonctions « paramétriques ». On se donne un espace de fonctions, comme les séries de Fourier ou une base de polynômes. Chacune des fonctions de cet espace est caractérisée par un ensemble fini et fixé de paramètres. Apprendre une telle fonction revient alors à apprendre ces paramètres ; ceux-ci sont appris à partir d'exemples. Pour cela on a recours une méthode des moindres carrés ou une descente de gradient.

Il existe également des approximateurs de fonctions non paramétriques : on ne se donne pas une forme de fonction prédéfinie comme ci-dessus, mais on se base uniquement sur les informations réellement connues au niveau des échantillons ; pour prédire la valeur en un point x, on recherche parmi les échantillons ceux qui ressemblent à x. On calcule à partir des ressemblances la valeur y de ce point [Loth *et al.*, 2007].

Dans le domaine de la planification, on peut avoir également recours à une approximation de la fonction de valeur quand l'espace d'états ou d'actions est relativement grand. Dans [Feng *et al.*, 2004], l'auteur utilise des fonctions constantes par morceaux ou linéaires par morceaux et continues. Dans [Pineau *et al.*, 2003], l'auteur reconstitue la fonction de valeur pour un POMDP en passant seulement par certains points clés de l'espace d'états. Récemment, [Hundt *et al.*, 2006] propose une approximation quadratique de la fonction de valeur pour un POMDP.

3.6 Décision multicritère et décomposition en parallèle

Il arrive parfois qu'une décision ne puisse pas être prise selon un seul et unique critère. La rationnalité, telle qu'elle a été définie précédemment, c'est-à-dire maximiser un critère unique n'est pas toujours possible. Imaginons par exemple un robot muni d'un seau rempli d'eau qui doit effectuer un parcours d'obstacle en un minimum de temps tout en gardant un maximum d'eau. Il est impossible dans la plupart des cas d'atteindre conjointement les deux objectifs.

La technique de décomposition en parallèle consiste dans un premier temps à décomposer un MDP global en sous processus, des sous-MDPs. Cependant, comme le montre schématiquement la figure 3.10, à chaque fois qu'une action sera appliquée, ce ne sera pas un seul sous-MDP qui sera affecté, mais l'ensemble de ceux-ci. L'action s'applique en parallèle sur chacun d'eux.

Pour donner une idée de la forme que peut prendre la décomposition en parallèle, on peut dire que les sous espaces des sous-MDPs forment une décomposition similaire à un produit cartésien du MDP de base, alors que c'est l'union de tous les sous-MDPs qui reforment le MDP de base dans la décomposition en série. Chaque sous-MDP peut avoir son propre espace d'actions, ou alors regrouper toutes les actions du MDP à décomposer. Pratiquement, l'idée clé de la décomposition

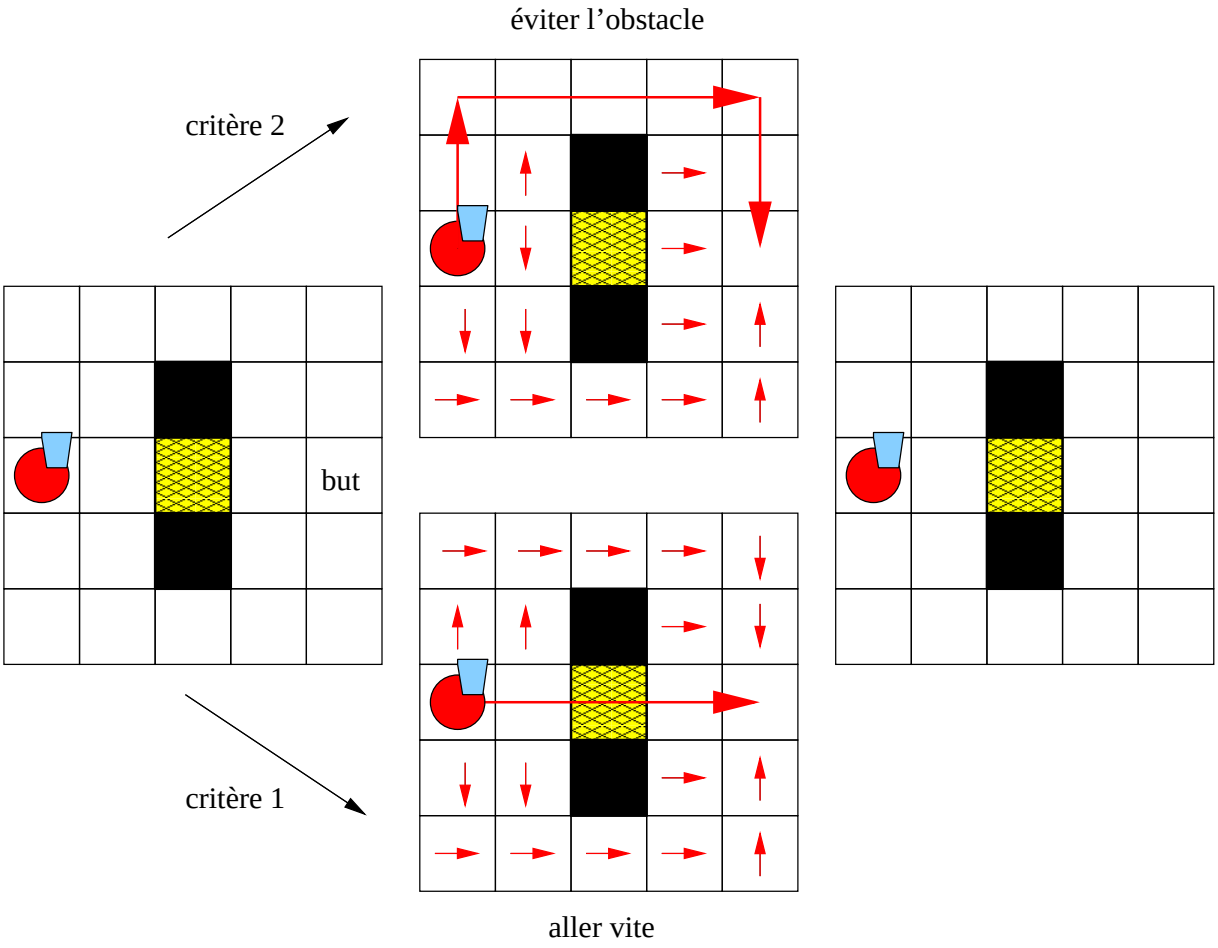


FIG. 3.10 – Décomposition en parallèle d'un MDP

explications *pour la figure 3.10 : le robot doit rejoindre la case but le plus rapidement possible en renversant un minimum d'eau. L'action qu'il effectuera à chaque instant aura un impact sur tous les critères. Prendre un virage le fait renverser de l'eau, mais peut le faire arriver plus rapidement que s'il traverse le bac à sable au milieu.*

en parallèle est de traiter chaque sous-processus sur un processeur indépendant. Mais exécuter une action sur un des sous-MDPs peut avoir des conséquences sur les autres MDPs, puisqu'une action a un effet sur chaque processus. Ils sont donc tous plus ou moins dépendants.

Dans le cadre de la décision multicritère, la décomposition en parallèle est plutôt adaptée. On peut penser par exemple à décomposer le MDP multicritère global en plusieurs sous-MDPs à qui l'on affecte séparément chaque critère. Ceux-ci se chargent ensuite chacun de leur côté d'optimiser leur critère [Mouaddib, 2006].

Singh et Cohn dans [Singh et Cohn, 1998] utilisent la décomposition en parallèle pour résoudre un problème de décision multicritère. Leur idée est de calculer une politique pour un MDP qui

optimise un des critères, puis de la recombiner avec un autre critère en éliminant au fur et à mesure les actions qui sont incompatibles, ainsi que les états qui ne sont plus accessibles.

Les auteurs de [Meuleau *et al.*, 1998] utilisent la même technique de décomposition dans un problème d'allocation de ressources. Ils compilent d'abord dans une phase "hors ligne" une politique pour chaque sous-MDP qui correspond à une macro-action. Ils réutilisent ensuite chacun des sous-MDP en ligne pour obtenir une décision convenable à chaque instant. Aucune politique globale n'est calculée explicitement.

S'inspirant du travail de [Mouaddib, 2004], M. Boussard [Boussard *et al.*, 2007] utilise la décision multicritère pour coordonner localement un ensemble d'agents. Dans un premier temps, chaque agent calcule un itinéraire sans tenir compte des autres, en minimisant le parcours total. Dans un deuxième temps, les agents excluent en ligne les actions qui gènent les autres agents, en essayant de ne pas trop s'éloigner de leur trajectoire de base. Il crée à l'exécution des ensembles d'actions qui gènent ou aident les autres agents, qui prennent finalement leur décision par rapport à une fonction de regret calculée localement et à cet instant pour la collectivité.

Nous avons exploité la technique de parallélisation pour décomposer un MDP à horizon fini dans [Mouaddib et Le Gloannec, 2006][25]. L'idée globale est de décomposer l'espace d'actions d'un MDP en sous espaces d'actions disjoints $\mathcal{A}_i$, de sorte que leur union forme l'espace d'actions $\mathcal{A}$ de départ. On crée ensuite un sous-MDP pour chaque sous espace d'actions $\mathcal{A}_i$ auquel on aura pris soin d'ajouter une action abstraite qui représentera une "autre action" d'un autre $\mathcal{A}_{j\neq i}$. Pour reconstruire chaque sous-MDP M_i, on génère l'espace d'état $\mathcal{S}_i$ grâce à un algorithme de type "recherche en avant". Chaque processeur P_i va ensuite calculer la valeur des états du sous-MDP M_i qu'il comparera avec les valeurs des états des autres sous-MDP $M_{j\neq i}$ qui seront transmises par communication. Les valeurs sont ensuite mises à jour dans chaque état en suivant le critère de maximisation de Bellman. Le temps de résolution est diminué grâce à la parallélisation des calculs, cependant le coût des communications de valeurs entre les différents processus amène un degré de complexité supplémentaire dans la résolution globale. On obtient ainsi un gain sous-linéaire. La technique de parallélisation est différente de celle énoncée en début de partie, mais les idées directrices restent globalement les mêmes.

Conclusion

Nous avons présenté un large panel de techniques utilisées pour résoudre un MDP de grande taille, c'est-à-dire avec un nombre d'états et/ou d'actions qui ralentissent la compilation de la politique obtenue, par le biais d'algorithmes classiques vus dans le chapitre précédent, comme value iteration. Des efforts de modélisation du MDP permettent de simplifier, de réduire la

[25] Ces travaux sont une extension à nos travaux de thèse, c'est pourquoi nous choisissons de ne pas y consacrer un chapitre entier.

représentation de celui-ci. Mais ces efforts ne suffisent pas toujours. Il faut faire appel à des techniques de calculs approchés de politiques, par abstraction par exemple. La décomposition permet quand à elle d'exploiter le MDP par régions et d'obtenir des politiques en recomposant les politiques locales. Beaucoup d'améliorations peuvent encore voir le jour sur ces nombreux sujets. On peut d'ailleurs penser à combiner toutes ces techniques en fonction du problème auquel on doit faire face.

Conclusion de la première partie

Certaines parties de l'univers sont encore inconnues et inaccessibles à l'homme, mais rien n'arrêtera son exploration. La robotique mobile et les systèmes embarqués offrent la possibilité de poursuivre l'aventure.

Cependant, dépourvue de mécanisme de raisonnement, une machine ne peut pas décider seule comment explorer ce monde que nous aimerions découvrir. Grâce aux recherches en intelligence artificielle, nous pouvons d'ores et déjà équiper des robots de tels mécanismes. Seulement, les résultats de la mission SOJOURNER, pendant laquelle le robot à passé 70% de son temps à ne rien faire, montrent à quel point les progrès en matière de planification *sous incertitudes* sont nécessaires. L'éloignement du robot à son contrôleur implique une grande part d'autonomie pour celui-ci, et l'environnement réel dans lequel il évolue apporte des incertitudes multiples : incertitudes temporelles, liées à la durée d'exécution des opérations et aussi aux événements asynchrones qui surviennent pendant la mission, incertitudes sur le résultat des opérations effectuées, incertitudes sur la réaction du robot face à certains événements, incertitudes face à la perception spatiale du monde...

Nous avons choisi de nous limiter aux incertitudes temporelles, et de centrer nos recherches vers le domaine de la planification sous incertitudes. Nous avons fait un tour d'horizon rapide sur l'histoire de la planification classique après quoi nous avons focalisé notre étude sur un outil fort intéressant pour modéliser les problèmes de planification sous incertitude : les processus décisionnels de Markov (MDP). Des algorithmes de programmation dynamique permettent de trouver une stratégie d'actions à partir du modèle MDP. On appelle ces stratégies "politiques", et ces algorithmes nous permettent d'en trouver une optimale parmi toutes. Malheureusement, malgré le confort que procure cet outil de modélisation, les problèmes de planification et de contrôle dans un monde réel sont composés d'une multitude de variables d'états, et les calculs de politiques deviennent souvent impossibles à mener.

C'est pourquoi nous avons présenté dans un dernier temps des méthodes qui permettent de trouver plus "intelligemment" des politiques qui approchent autant que faire se peut l'optimale, en restructurant les données (nouvelle modélisation de la fonction de transition qui modélise l'incertitude, factorisation des variables qui constituent l'état). En abstrayant le problème, c'est

à dire en réduisant le modèle en regroupant certains états entre eux, on peut diminuer le temps de calcul pour obtenir une politique presque optimale. Dans le même ordre d'idée, la décomposition est une technique qui consiste à séparer le calcul sur plusieurs sous parties du MDP que l'on scinde en sous-MDPs, basé sur le principe bien connu *"diviser pour régner"*.

Cette première partie regroupe des outils pour modéliser l'incertitude en planification. Cependant, nous avons également besoin de gérer les ressources limitées de notre robot, et nous allons voir dans le chapitre suivant comment le raisonnement progressif nous permet de tenir de la ressource temporelle. Nous allons également montrer qu'il est possible d'étendre ce mode de raisonnement à plusieurs ressources consommables. Ensuite, nous nous pencherons sur les problèmes qui découlent directement de l'arrivée d'événements asynchrones pendant la phase d'exécution du plan, et comment calculer dynamiquement une politique *correcte* pour faire face à l'imprévu. Cette partie sur la planification dans un environnement dynamique sera suivie d'une dernière partie dans laquelle nous validerons nos résultats par le biais de plusieurs expérimentations.

Deuxième partie

Contrôle du raisonnement progressif

Chapitre 4

Introduction au raisonnement progressif

*It appears probable that, however adaptive the behavior of organisms in learning and choice situations, this adaptiveness falls far short of the ideal "**maximizing**" postulated in economic theory. Evidently, organisms adapt well enough to "**satisfice**"; they do not, in general, "**optimize**".*

- Herbert Simon -

Introduction

Dans notre contexte, un robot va effectuer une séquence de tâches complexes. Les ressources consommables embarquées sont limitées. Le temps pour effectuer la mission est limité, l'espace mémoire pour stocker les vidéos et les photos aussi. Notre motivation première est de pouvoir permettre au robot de contrôler l'exécution ces tâches, afin qu'il puisse adapter sa consommation de ressources sans forcément obtenir de chaque exécution locale un résultat optimal pour cette tâche particulière, mais obtenir une solution globale satisfaisante pour l'ensemble de la mission.

Le raisonnement flexible, que nous allons présenter dans ce chapitre, s'appuie sur la constatation d'Herbert Simon : dans la nature, les êtres vivants s'adaptent aux situations jusqu'à ce qu'il soient satisfaits, ils n'optimisent généralement pas le résultat de leur comportement.

Nous allons expliquer dans ce chapitre comment le raisonnement flexible va nous permettre d'établir un mécanisme de contrôle similaire au mécanisme naturel présenté par H. Simon, afin que le robot soit capable de juger à tout moment si la façon d'exécuter une tâche est opportune vis à vis des ressources qu'il lui reste. Plus synthétiquement, le robot doit adapter sa consommation de ressources, afin de maximiser le résultat final de la mission.

Après avoir présenté les enjeux du raisonnement flexible, nous présenterons plusieurs systèmes de contrôle adaptatifs : contrôle d'un algorithme anytime, le modèle Design to Time, et finalement, le raisonnement progressif, que nous avons finalement retenu.

L'élaboration de tels systèmes fonctionne toujours en deux temps :

- la conception d'un modèle de raisonnement
- le mécanisme de contrôle adaptatif

C'est après avoir motivé notre choix que nous exposerons plusieurs **applications potentielles**, dont le contrôle robotique. Le chapitre suivant sera quant à lui consacré à la formalisation du raisonnement progressif.

4.1 Le raisonnement flexible

4.1.1 Pourquoi le raisonnement flexible

Le raisonnement flexible se base sur la constatation suivante :

Les ressources nécessaires à la construction d'une décision optimale réduisent l'utilité globale du système

L'exemple tragique d'un robot sauveteur en cas d'avalanche ou de séisme nous montre bien que la construction d'une politique optimale devient inutile si un blessé perdu meurt pendant que le robot réfléchit au chemin qu'il va prendre pour le retrouver. Ici, notre ressource est le temps, la qualité peut être une mesure sur le chemin que le robot va emprunter[26], et l'utilité est l'état dans lequel sera notre blessé au moment où le robot le rejoindra.

Le raisonnement flexible est un moyen de construire une solution utile, non nécessairement optimale, et de s'adapter à la situation dans laquelle le système se trouve. On s'en sert dans les cas suivants :

- le calcul de la solution optimale est infaisable, par manque de ressources (mémoire, processeur...)
- il n'est pas souhaitable (dans un certain contexte) de construire la solution optimale.

L'**utilité** d'une solution est un **compromis** entre la **qualité** de celle-ci et les **ressources** utilisées pour l'obtenir.

L'élaboration de systèmes flexibles fonctionne toujours en deux temps :

- la conception d'un modèle de raisonnement
- le mécanisme de contrôle adaptatif

La conception du modèle doit permettre de pouvoir déployer des algorithmes adaptés au problème à traiter, de façon à ce que l'agent puisse contrôler l'utilité de son comportement.

Il existe déjà plusieurs familles de systèmes de raisonnement flexible connus parmi lesquels nous avons : le contrôle des algorithmes anytime, celle des méthodes multiples, et finalement celle du raisonnement progressif à laquelle nous nous sommes plus particulièrement intéressés.

[26] la longueur du chemin, les risques pris pendant le trajet...

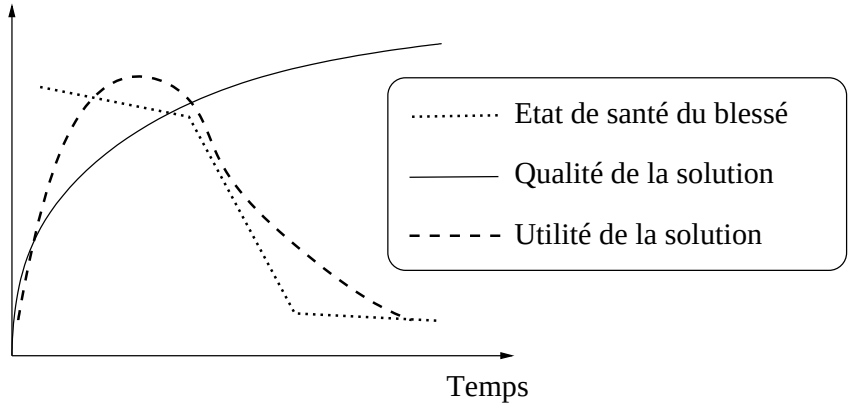


FIG. 4.1 – L'utilité d'une solution

Nous avons représenté sur la figure 4.1 trois courbes dont l'aspect représente respectivement l'état d'un blessé à sauver, la qualité du chemin calculée par le robot (d'un algorithme permettant de trouver le plus court chemin par exemple), et en pointillés l'utilité de d'obtenir la solution en fonction du temps. La solution d'utilité maximale ne correspond pas au moment où le robot trouve le chemin le plus court, il faut tenir compte de la contrainte du problème, c'est-à-dire ici, l'état du blessé.

4.1.2 Algorithmes anytime

Les algorithmes anytime sont des algorithmes capables de retourner une solution "*at any time*", c'est à dire à n'importe quel moment. Ils sont introduits par T. Dean et M. Boddy dans [Dean et Boddy, 1988]. La solution retournée s'améliore dans le temps. On se donne une mesure de qualité de cette solution. Le profil de qualité ou "*performance profile*" est la fonction permettant de décrire le comportement de l'algorithme anytime (voir figure 4.2).

S. Zilberstein a décrit dans [Zilberstein, 1996] les propriétés que doit avoir un bon algorithme anytime :

1. **Interruptibilité** : l'algorithme peut être interrompu à n'importe quel moment et doit fournir une solution.
2. **Monotonie** : la qualité est une fonction croissante du temps.
3. **Qualité mesurable** : la qualité des solutions approximatives doit être définie précisément.
4. **Amélioration descendante** : l'amélioration de la qualité de la solution doit décroître au cours du temps (la dérivée du profil de qualité est décroissante).
5. **Consistance** : Pour un même temps alloué à un même algorithme, la qualité est la même.
6. **Qualité identifiable** : la qualité d'une solution intermédiaire doit être calculée facilement.

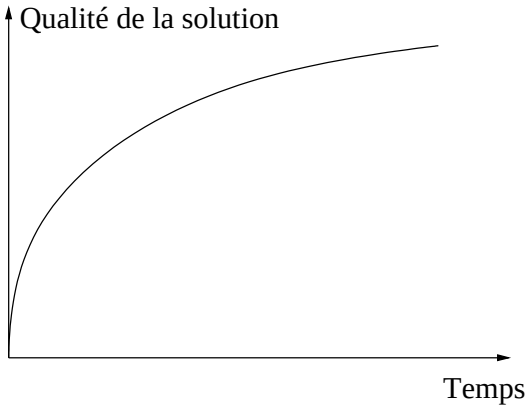


FIG. 4.2 – Un bon profil de qualité pour un algorithme anytime

7. **Préemptibilité** : l'algorithme doit pourvoir être suspendu et repris[27] sans grande perte.

On retrouve de nombreux travaux se basant sur les algorithmes anytime ; sur les systèmes temps réels, dans [Zilberstein et Russell, 1996], pour la planification dans [Knoblock, 1994, Boddy et Dean, 1994], le diagnostic [Ash *et al.*, 1993, Pos, 1993, Harlemen et Teije, 2000], la recherche heuristique [Korf, 1987], le contrôle [Mouaddib, 1993, Mouaddib et Zilberstein, 1995], et enfin l'ordonnancement [Boddy et Dean, 1994, Mouaddib et Zilberstein, 1997].

Le mécanisme de contrôle des algorithmes anytime se fonde sur une fonction d'utilité qui permet d'indiquer le moment où la solution (même si elle n'est pas optimisée) doit être retournée (comme sur la figure 4.1) .

4.1.3 Méthodes multiples

Avec cette approche, on suppose que l'on connaît un ensemble de méthodes réalisant toutes un compromis précis entre la qualité de la solution et la durée d'exécution au pire cas pour un problème donné. On peut alors choisir parmi ces différentes méthodes de façon à résoudre le problème tout en respectant le compromis. Il existe notamment le modèle TAEMS [Decker et Lesser, 1993] puis le "Design-To-Time" [Garvey et Lesser, 1993]. Dans TAEMS, chaque tâche est divisée en un graphe acyclique de sous-tâches pour lesquelles on dispose d'opérateurs pour recomposer la qualité globale de la tâche. "Design-To-Time" se base sur des algorithmes de recherche opérationnelle, avec un ensemble de tâches définies comme dans TAEMS, avec une composante permettant de superviser l'incertitude de la qualité de la solution produite. Plus le temps alloué pour ces algorithmes est grand, meilleure sera la solution.

[27] on doit pouvoir recommencer là où on en était.

4.1.4 Le raisonnement progressif

Le raisonnement progressif s'inscrit dans la même mouvance que les algorithmes anytime, à savoir que la qualité du résultat obtenu augmente en fonction des ressources allouées. Nous allons dans un premier temps présenter le modèle de représentation des tâches contrôlées par le raisonnement progressif. Dans le chapitre suivant, nous formaliserons et présenterons des algorithmes de contrôles dédiés.

Le modèle a connu une évolution au cours du temps, pour s'adapter à de nouveaux problèmes. La toute première approche est hiérarchique, on effectue chaque tâche niveau par niveau. Le deuxième modèle est hiérarchique et modulaire, permettant d'adapter la façon dont est exécuté chaque niveau.

4.1.5 L'approche hiérarchique

Elle se base sur un raisonnement hiérarchisé par niveau. Les niveaux les plus bas concernent les connaissances basiques du problème, qui sont essentielles à la construction de la solution. Les niveaux supérieurs contiennent des informations moins importantes qui permettent d'affiner celle-ci. Plus on exploite les hauts niveaux, plus la solution sera précise, moins on les exploite, plus la solution sera obtenue rapidement. Cette approche permet donc de construire rapidement une solution approximative à un problème que l'on peut améliorer **progressivement**. La hiérarchisation des niveaux et la qualité qui doit en ressortir sont autant de données qui doivent être décrites par l'expert qui pose le problème à résoudre (dans notre cas, les contrôleurs du robot).

Le raisonnement progressif possède des propriétés similaires aux algorithmes anytime :

- Interruptibilité : on peut arrêter l'algorithme à tout moment, il retournera la solution du dernier niveau effectué.
- Monotonie : une amélioration effectuée après un niveau supplémentaire augmente la qualité globale de la solution
- Amélioration décroissante : les premières améliorations sont faites dans les premiers niveaux, les derniers niveaux améliorent peu la solution. L'importance des données décroît avec les niveaux.

Une solution peut donc être obtenue à la fin de chaque niveau, le raisonnement progressif peut être considéré comme une discrétisation des algorithmes anytime.

On peut, grâce au raisonnement progressif, contrôler la qualité de la solution obtenue en fonction des ressources consommées pour l'obtenir. L'interruptibilité assure de pouvoir abandonner à tout instant l'exécution d'une tâche complexe, de façon à garder des ressources pour plus tard.

Ce premier modèle a été utilisé pour mettre au point un système de contrôle ferroviaire grâce au modèle GREATS[Mouaddib, 1993, Mouaddib *et al.*, 1994].

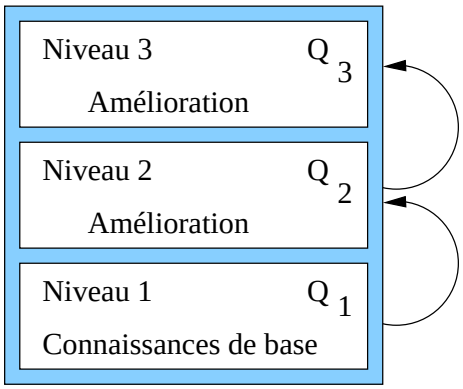


FIG. 4.3 – Le raisonnement progressif : approche hiérarchique

Sur la figure 4.3, nous avons représenté une unité de raisonnement progressif hiérarchique. Les connaissances nécessaires à la résolution d'un problème sont stockées dans le niveau 1. *Deux améliorations successives sont possibles. La qualité des niveaux successifs est décroissante :* $Q_1 > Q_2 > Q_3$. *La qualité finale de la solution peut être soit* Q_1, *soit* $Q_1 + Q_2$, *soit* $Q_1 + Q_2 + Q_3$.

4.1.6 L'approche modulaire

Pour représenter l'exécution de tâches complexes comme celles qui vont être effectuées par le robot par exemple, l'approche hiérarchique seule ne suffisait plus. Le modèle a donc été étendu à une approche modulaire, qui offre la possibilité de choisir la façon dont sera exécuté chaque niveau.

À chaque niveau, l'agent exécute un seul module parmi tous ceux qui lui sont proposés dans ce niveau. Celui-ci peut également interrompre l'exécution de la tâche après un niveau donné, ou continuer d'effectuer cette tâche.

La qualité de la solution obtenue sera la somme des qualité des modules choisis. Ceci permet d'avoir une plus grande combinaison de résultats obtenus en sortie, ainsi qu'une plus grande combinaison de consommation totale de ressources possible. La modularité introduit un nouveau degré d'adaptation du nombre de ressources à consommer vis à vis du résultat à obtenir.

Remarque : la qualité obtenue ne donne que la qualité du résultat obtenue localement. Elle ne mesure en aucun cas l'utilité d'avoir obtenu cette qualité. C'est la fonction d'utilité qui permettra de contrôler si la qualité obtenue présente un résultat intéressant, connaissant les ressources consommées[28].

Remarque : ce ne sont pas les algorithmes qui permettent d'obtenir une politique à partir du modèle du raisonnement progressif qui sont flexibles (ces algorithmes ne sont **pas** des algorithmes anytime) mais la politique obtenue par cet algorithme qui permettra à l'agent d'adopter un

[28] L'explication du mécanisme du contrôle par la fonction d'utilité est donné dans le chapitre suivant.

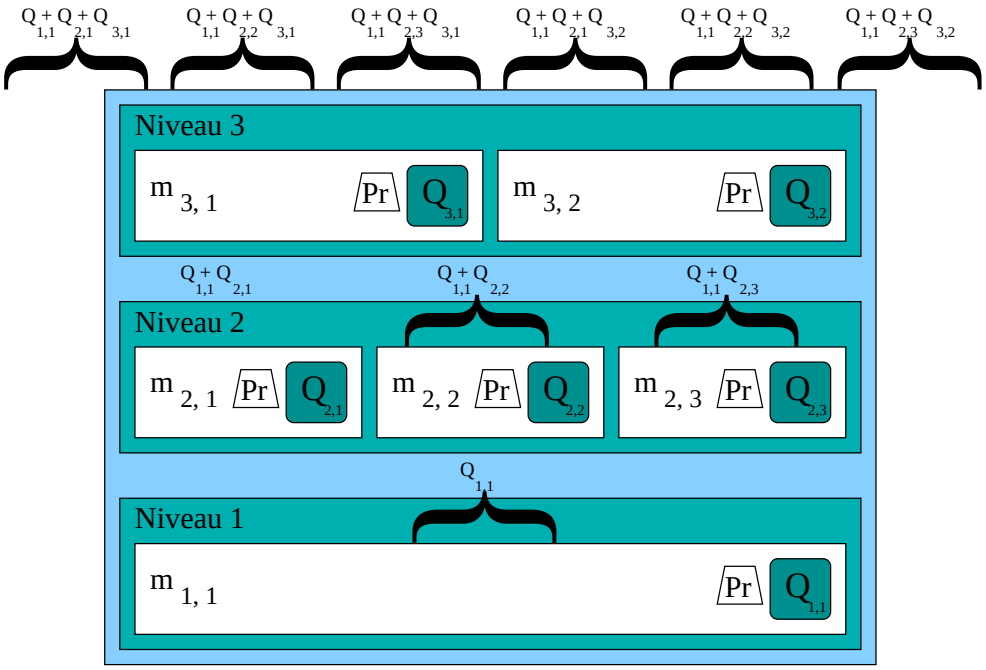


FIG. 4.4 – Une unité raisonnement progressif modulaire.

Sur la figure 4.4, nous avons représenté une unité de raisonnement progressif décrivant une tâche complexe hiérarchique et modulaire : l'agent l'effectuera de bas en haut, en ayant la possibilité de s'arrêter après chaque niveau. Effectuer un niveau consiste à choisir un module, puis à l'exécuter. L'exécution de chaque module produit une certaine qualité, et consomme une certaine quantité de ressources. le module $\mathsf{m}_{2,1}$ *est le premier module du deuxième niveau, il est une façon de faire le niveau* 2*, et il produit une qualité* $\mathbf{Q}_{2,1}$ *inférieure à* $\mathbf{Q}_{2,2}$*. En sortie on peut obtenir toutes les combinaisons de qualités indiquées sur la figure.*

comportement dont le contrôle sera flexible. Il faut bien distinguer le modèle de raisonnement du mécanisme de contrôle.

La qualité du résultat obtenu en sortie d'une unité de raisonnement progressif n'est validée que si le dernier niveau a été atteint. On introduit parfois un module *sauter*, qui permet de sauter le niveau sans l'exécuter. Ce module ne consomme pas de ressources et son exécution produit une qualité nulle.

Le raisonnement progressif trouve déjà plusieurs cadres applicatifs que nous allons décrire ci-après.

4.2 Cas d'utilisation du raisonnement progressif

En 1997 - 1998 , A.I. Mouaddib et S. Zilberstein décrivent dans [Mouaddib et Zilberstein, 1997, Mouaddib et Zilberstein, 1998] le formalisme du raisonnement progressif. De nombreux travaux d'approfondissement apparaissent ensuite, essentiellement encadrés par les deux auteurs. Nous allons présenter deux contextes applicatifs du raisonnement progressif. A. Arnt et S. Zilberstein et A.I. Mouaddib ont tout d'abord appliqué le raisonnement progressif à un moteur de recherche sur internet. S. Cardon, A.I. Mouaddib et S. Zilberstein l'ont quant à eux adapté à la robotique mobile.

4.2.1 Un moteur de recherche

Un moteur de recherche est une application qui, comme son nom l'indique, permet à un utilisateur connecté à internet de retrouver certains documents spécifiques sur la toile en fonction de certains critères ou mots clefs. Il existe déjà plusieurs sites très connus qui proposent ce service aux internautes. L'utilisation est simple : il suffit d'envoyer donc une requête à ce moteur (souvent une liste de mots clefs) qui renvoie une liste de liens vers des documents. Leur contenu est en relation plus ou moins pertinente avec la requête de l'utilisateur. Celui-ci devra choisir parmi les liens qui lui sont proposés les documents auxquels il accédera.

Les documents qui sont retournés ne correspondent pas forcément à l'attente du client. On pourrait imaginer un algorithme plus "fin" pour l'exploitation des mots clés de la requête qui permettrait de renvoyer des résultats plus pertinents. Un algorithme certes plus lent, mais dont les résultats seraient meilleurs. Malheureusement, l'affluence de requêtes sur ce type de serveur à certaines heures entraîne une charge de calcul pour les réponses à fournir, et on va alors privilégier les algorithmes de recherches simples et rapides pour faire face à ce phénomène de surcharge.

L'idée des auteurs de [Zilberstein *et al.*, 2000] est de contrôler cette charge de requêtes afin d'adapter la qualité des réponses. Quand beaucoup de requêtes arrivent en même temps, la recherche sera rapide, et la réponse de moins bonne qualité, si peu de requêtes arrivent au même moment, une recherche plus approfondie pourra être effectuée, satisfaisant ainsi l'utilisateur. Ils associent à chaque requête une unité de raisonnement progressif qui correspond aux différentes manières possibles de traiter celle-ci. Un traitement simple donnera une solution de faible qualité, et vis-versa. Une fonction d'utilité $\mathbf{U}(\mathbf{Q}, \mathtt{t}) \rightarrow \mathbb{R}$ permet de mesurer l'utilité d'obtenir une qualité $\mathbf{Q}$ au bout d'un temps $\mathtt{t}$ de traitement. Plus la qualité de la solution augmente, plus l'utilité augmente, mais plus le temps écoulé augmente, plus l'utilité diminue. Ainsi, les utilisateurs seront satisfaits puisque d'un côté, leur temps d'attente sera pris en compte, de l'autre la qualité de la réponse ne sera pas négligée.

Ainsi il y a trois façons d'accomplir le niveau de base de l'unité de raisonnement progressif du moteur de recherche, et deux façons d'accomplir le second. Dans le premier niveau, une

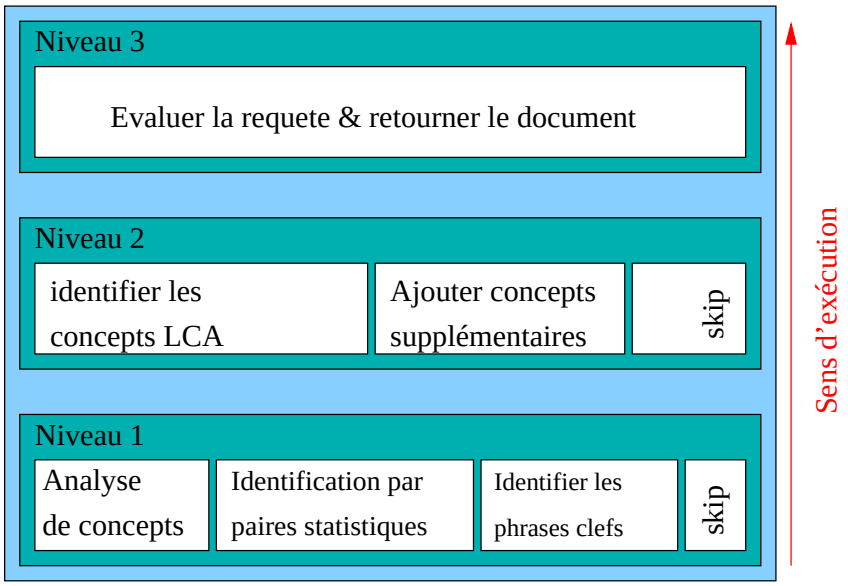


FIG. 4.5 – Une unité de raisonnement progressif typique pour le moteur de recherche.

La figure 4.5 présente une unité de raisonnement progressif pour le moteur de recherche adaptatif d'après [Mouaddib et Zilberstein, 1999]. Les modules représentent des méthodes de recherche de documents typiques à ce domaine d'application.

recherche de concepts simples dans la requête avec le module 1 est effectuée, ou alors une analyse plus précise à l'aide de paires statistiques dans le module 2, ou une identification de phrases clefs le module 3. On peut aussi *sauter* le niveau, c'est à dire ne rien faire. Rappelons qu'à chaque niveau, il faut « choisir » un seul module à exécuter. Nous avouons humblement ne pas être spécialiste de la linguistique ni de la recherche d'informations, et les concepts introduits dans cette tâche complexe ne sont pas des éléments que nous maîtrisons parfaitement. Ceci-dit, l'exécution du module 3 prendra certainement plus de temps que les deux autres, mais renverra un résultat de meilleur qualité. Il est ensuite possible d'améliorer le résultat de la requête grâce à un des deux modules du niveau 2. On peut également *sauter* ce niveau : si la charge du serveur est importante, celui-ci ne jugera pas utile de continuer à améliorer cette solution et renverra donc le résultat tel qu'il a été trouvé à la fin du premier niveau.

On peut de cette manière adapter intelligemment la qualité de la réponse à la requête d'un utilisateur en fonction de la quantité de requêtes en charge sur le serveur.

Tous ces travaux ont été essentiellement réalisés à l'université UMASS de Amherst, Massachussets (USA) et le LORIA de Nancy par A. Arnt et S. Zilberstein, A. I. Mouaddib et F. Charpillet.

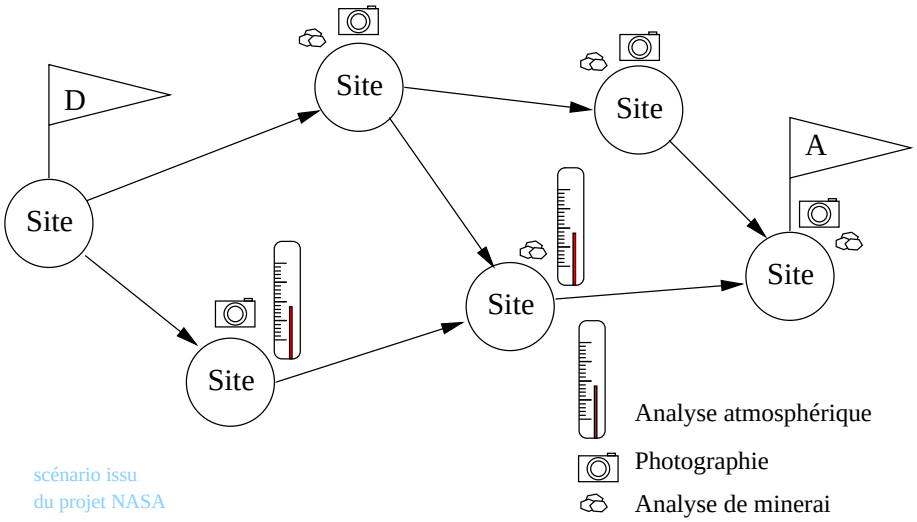


FIG. 4.6 – Un graphe de mission type

Explications *pour la figure 4.6 : le robot a une mission : explorer certains sites (noeuds) de ce graphe, dans un ordre imposé par ce graphe acyclique. Il partira du site de départ marqué d'un* ***D*** *et arrivera, si le temps le lui permet, au site d'arrivée marqué d'un* ***A****. Il passera plus ou moins de temps sur chaque site pour récolter des informations, à lui de juger ce qui doit être fait, à nous de lui donner les moyens de décider.*

4.2.2 En robotique mobile

Dans un cadre différent, nous essayons d'adapter le raisonnement progressif au contrôle d'un robot mobile. Ces travaux ont été amorcés dans le cadre du projet national ROBEA[29] par A. I. Mouaddib et F. Charpillet. Comme nous l'avons précisé au début de ce mémoire, le but principal de notre étude est de doter une entité robotique d'un mécanisme de décision afin qu'elle puisse accomplir, sans aide extérieure, certaines missions dans des endroits encore inaccessibles à l'homme. Le cas d'un robot explorateur autonome évoluant sur la planète Mars illustre parfaitement ce but. On donne à ce robot un ensemble de sites à visiter. Le sens de parcours possible de ces sites est sous forme de graphe acyclique, on l'appelle mission(voir figure 4.6). Le robot devra alors visiter certains sites, et y effectuer des prélèvements (prélèvements atmosphériques, de minerai au sol, prise de photographie etc...) donc passer plus ou moins de temps sur chaque site.

A tout cela se rajoute un certain nombre de contraintes : les communications entre la planète Mars et la terre sont difficiles et le robot ne pourra transmettre les résultats de ses analyses qu'à un endroit donné, le site d'arrivée, et dans une fenêtre de temps précise, au delà de laquelle toute communication avec la planète Terre risque d'être compromise.

[29] Projet en partenariat avec l'équipe MAD du GREYC de Caen et l'équipe MAIA du LORIA de Nancy.

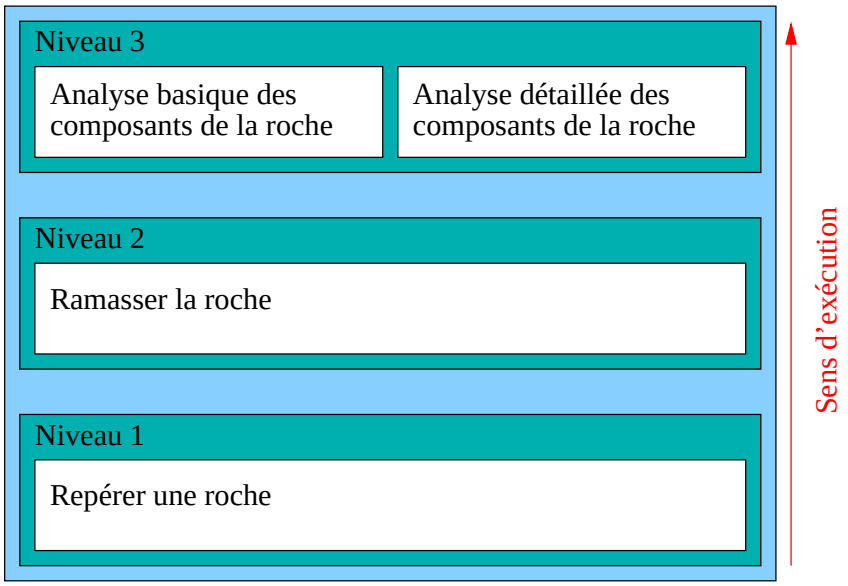


FIG. 4.7 – Une tâche typique à réaliser par un robot explorateur.

Explications *pour la figure 4.7 : le robot explorateur récolte des informations un site : les scientifiques voudraient qu'il fasse une analyse chimique précise d'une roche. Cette analyse doit se dérouler en trois étapes : d'abord le robot doit repérer la roche, ensuite il doit la ramasser, puis finalement, l'analyser avec différents outils. Il peut analyser rapidement cette roche, avec des capteurs basiques, ou faire une analyse plus précise. L'unité de raisonnement progressif décrit l'ordre dans lequel ces tâches doivent être effectuées.*

Techniquement, le robot reçoit en début de journée une mission, avec les sites à visiter. Il doit également profiter du soleil pendant une partie de la journée pour recharger ses batteries[30]. Une fois arrivé au site final, il renvoie à la Terre les informations qu'il a récoltées, puis reçoit une nouvelle mission. Ne pas arriver au site final est donc synonyme de perte d'informations, celle de la mission du jour ne sont pas transmises et la mission du lendemain est annulée. Dans le pire des cas, le robot se perd et ne recharge pas ses batteries, c'est un échec total.

Le temps nécessaire pour effectuer chaque mission est donc limité. D'un côté, plus le robot fera d'analyses, plus les scientifiques qui attendent ces résultats seront satisfaits. Si le robot n'atteint pas le site d'arrivée à la date escomptée, ou s'il n'y arrive jamais, la mission sera considérée comme un échec puisque les scientifiques n'auront aucun retour. Si le robot arrive au site d'arrivée bien avant l'échéance, les scientifiques vont juger que le robot aurait pu récolter plus d'informations et vont être également déçus. Il va donc falloir un moyen efficace pour trouver un compromis entre la qualité des analyses faites sur ces sites et le temps passé pour le faire.

Chaque relevé est suivi d'une analyse. Le robot peut par exemple ramasser du minerai au

[30] Le robot possède des panneaux solaires pour charger ses batteries

sol, puis analyser les matériaux qui le compose. Puisqu'une analyse ne peut se faire sans relevé, mais qu'un relevé sans analyse peut être malgré tout considéré comme un résultat intéressant pour les scientifiques, le raisonnement progressif est un mécanisme de contrôle intéressant pour ce type de mission. La figure 4.7 illustre une des tâches que le robot pourrait effectuer sur un des sites.

Dans [Cardon *et al.*, 2001], les auteurs utilisent le formalisme du raisonnement progressif pour décrire une mission d'exploration, et ils proposent un moyen de calculer une politique de façon à ce que le robot puisse décider à tout moment s'il doit récolter des données ou avancer vers le site suivant. Dans cet article, les auteurs se sont donnés une contrainte supplémentaire. Ils ont considéré que sur chaque site, les relevés devaient être faits avant une date butoir, au delà de laquelle tout relevé serait impossible.

C'est cette application à la robotique mobile qui a le plus retenu notre attention au cours de notre étude.

Conclusion

Ce chapitre présente le raisonnement progressif et deux applications potentielles : un moteur de recherche et un robot autonome explorateur. Nous basons notre présentation sur le fait que dans certaines applications, les ressources nécessaires à la construction d'une solution optimale réduisent l'utilité globale du système.

Nous présentons dans ce chapitre le raisonnement flexible grâce auquel il est possible de concevoir des architectures de contrôle d'exécution de tâches complexes permettant de construire pas à pas des solutions en adaptant la quantité de ressources consommées au problème à traiter.

La construction de telles architectures se passe en deux temps :

- la modélisation du problème à résoudre, d'un modèle de raisonnement,
- l'élaboration d'un mécanisme de contrôle adaptatif.

Le raisonnement anytime est une architecture de raisonnement flexible qui permet d'adapter la qualité de la solution d'un problème donné en fonction du temps que l'on peut allouer à cette résolution. D'un coté, un algorithme anytime calcule une solution à un problème et garantit de renvoyer une solution (de qualité croissante avec le temps passé) quelque soit le temps au bout duquel l'algorithme s'arrête, de l'autre, une fonction d'utilité contrôle l'exécution de cet algorithme de façon à renvoyer un résultat au "bon moment".

Le raisonnement progressif permet quant à lui :

- De modéliser une tâche complexe en la divisant en étapes appelées niveaux, chaque niveau pouvant être subdivisé en modules,
- l'élaboration d'un mécanisme de contrôle adaptatif, en analysant le compromis entre la qualité obtenue en sortie de la tâche effectuée et les ressources dépensées pour obtenir

cette qualité.

Le raisonnement progressif garantit une double adaptation :

- l'approche modulaire permet d'adapter la consommation de ressources à chaque niveau,
- l'approche hiérarchique permet d'interrompre la tâche à n'importe quel moment pour économiser les ressources pour la suite.

Ainsi, ce mécanisme permet, si l'agent à contrôler dispose d'une mesure de l'utilité de la dépense de ressources pour une tâche donnée en fonction de la qualité obtenue, d'adapter la consommation de ressources, pour ne pas forcément rechercher un résultat optimal, mais un résultat satisfaisant.

Dans le chapitre suivant, nous allons voir comment formaliser le raisonnement progressif et le mécanisme de contrôle adaptatif. Nous allons présenter les algorithmes qui permettront de calculer une politique qui permettra au robot d'avoir un comportement utile vis à vis des résultats des tâches et des ressources consommées.

Chapitre 5

Approches de contrôle en raisonnement progressif

Introduction

Notre approche de contrôle pour un agent autonome se veut flexible et nous avons choisi de la fonder sur le raisonnement flexible et plus particulièrement le raisonnement progressif, dont nous avons présenté les fondements dans le chapitre précédent. Ce chapitre est divisé en trois parties :

- la **formalisation** de la mission et des unités de raisonnement progressif,
- la description de **deux algorithmes** permettant de calculer la fonction de valeur et la politique pour toute la mission,
- un point de vue différent, le **coût occasionné**, qui permet de focaliser le contrôle à la tâche courante en fonction du reste de la mission.

Le formalisme du problème sera présenté hiérarchiquement : la mission est découpée en unités de raisonnement progressif (PRUs), elles-mêmes divisées en niveaux. Nous introduisons de nombreux indices pour décrire le modèle, qui peut rendre sa lecture légèrement fastidieuse. Nous nous tiendrons à ces notations pour toute la suite du mémoire.

Rappelons que l'agent est rationnel. Nous présenterons comment celui-ci va optimiser son critère de gain. Nous verrons que la complexité liée au calcul de la fonction de valeur est proportionnelle à la taille de l'espace d'états. Nous allons donc procéder à un dénombrement de cet espace d'états pour une mission donnée.

Le calcul global de la politique et de la fonction de valeur associée peut se faire via deux algorithmes de programmation dynamique menant tous deux vers une politique identique. L'algorithme par programmation dynamique classique explore, construit et évalue récursivement chaque état (avec mémoization), tandis que l'algorithme **par niveaux** construit exhaustivement

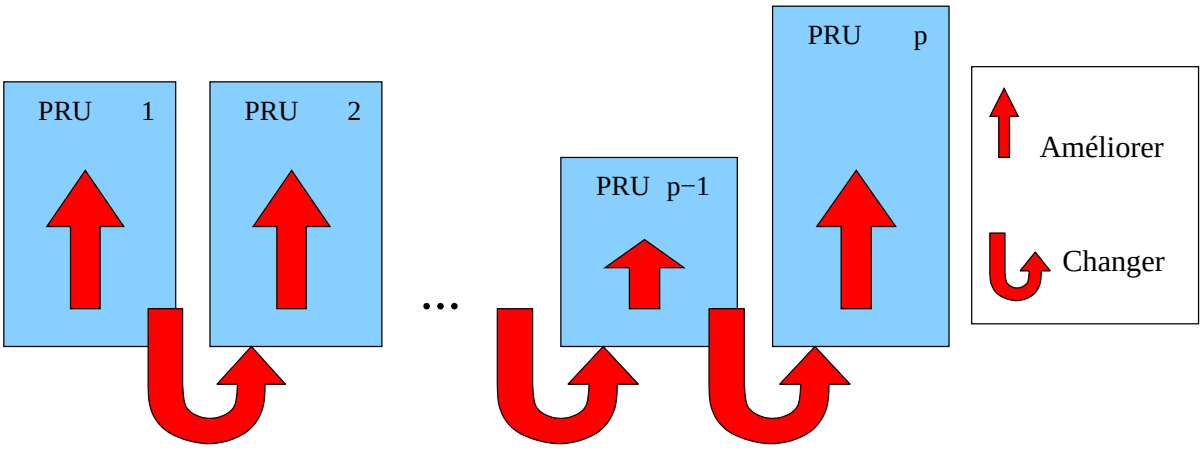


FIG. 5.1 – Une mission

Pendant une mission, l'agent peut améliorer progressivement chaque PRU en exécutant les niveaux internes, ou la quitter pour la PRU suivante. Il ne peut jamais revenir sur ses pas.

l'ensemble des états possibles, et les évalue par chaînage arrière.

Le coût occasionné offre un point de vue différent, et permet toujours d'obtenir un contrôle optimal pour l'agent rationnel. Sans réellement diminuer la complexité globale du système, le coût occasionné permet de séparer le calcul de la politique de la tâche courante de l'évaluation de la fonction de gain espéré pour le futur. Nous utiliserons ce point de vue plus tard, lorsque nous présenterons notre méthode de planification rapide dans le chapitre 7.

5.1 Formalisme

Nous allons définir un par un les concepts qui décrivent le modèle du raisonnement progressif, en commençant par celui qui encapsule tout (la mission) jusqu'aux plus fins (les ressources).

Une **mission** est une séquence finie d'unités de raisonnement progressif (notées PRU). Nous attribuons un indice unique $\mathtt{p}$ à chaque $\mathsf{PRU}_{\mathtt{p}}$, $\mathtt{p} \in [1, ..., \mathbf{P}]$ (voir figure 5.1).

note : nous pourrions étendre la définition de mission à un graphe acyclique de PRUs. Une liste de PRUs est un chemin possible dans un graphe acyclique de PRUs. Nous simplifions donc le problème pour des raisons de clarté d'expression, car les indices utilisés pour décrire le modèle sont nombreux. Le passage au graphe acyclique pose le problème du choix du meilleur chemin, mais nous nous contenterons pour cette thèse d'un seul chemin possible.

Une **unité de raisonnement progressif** est une séquence finie de $\mathtt{N}$ **niveaux** que l'agent doit exécuter dans l'ordre. Nous nous basons sur la représentation de la figure 5.2 pour cette définition. L'agent ne peut pas revenir sur un niveau déjà exécuté. À la fin d'un niveau l'agent peut décider d'exécuter ou non le niveau suivant. Il existe une contrainte de précédence entre

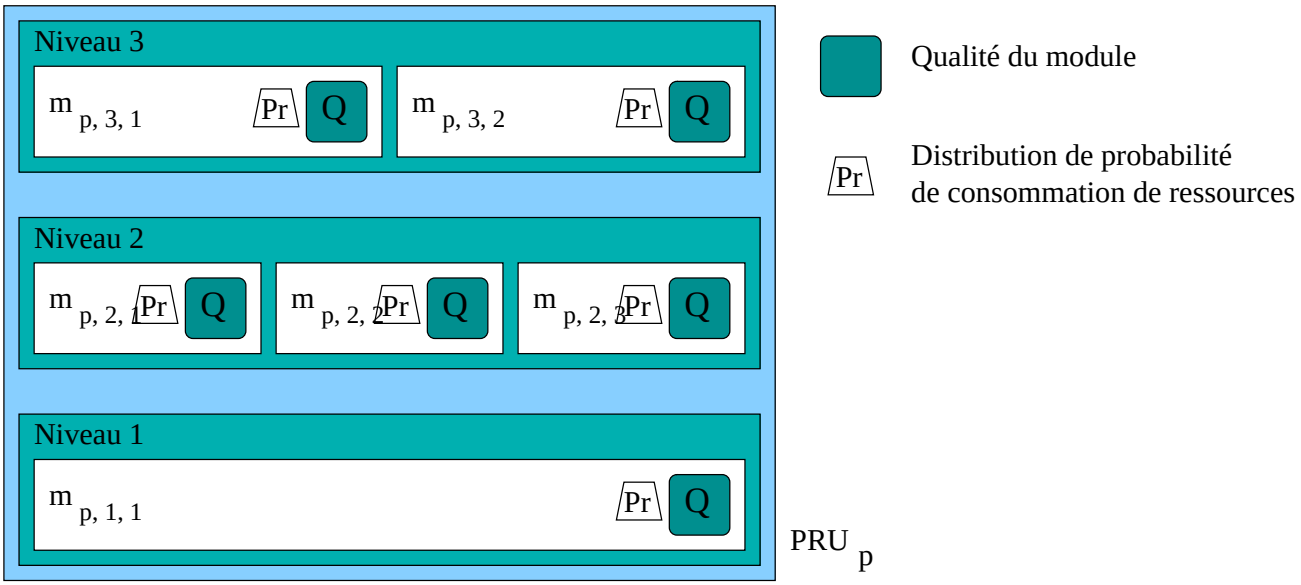


FIG. 5.2 – Définition d'une unité de raisonnement progressif

Cette figure est une PRU formalisée. Elle peut représenter n'importe quelle type tâche progressive, comme celle que nous avons proposée pour le robot explorateur (figure 4.7) ou le moteur de recherche (figure 4.5), le module $\mathsf{m}_{\mathtt{p},3,1}$ *n'a pas de nom, c'est le premier module du troisième niveau de la* $\mathsf{PRU}_{\mathtt{p}}$

les niveaux : le niveau $\mathtt{n}+\mathtt{1}$ ne peut commencer que si le niveau $\mathtt{n}$ a produit un résultat et est terminé. S'il n'exécute pas le niveau suivant, alors l'agent arrête d'exécuter cette PRU en entier. Nous noterons $\mathsf{N}_{\mathtt{p},\mathtt{n}}$ le $\mathtt{n}^{\text{ème}}$ niveau de la $\mathsf{PRU}_{\mathtt{p}}$. Si le dernier niveau de la PRU n'est pas exécuté, elle ne rapporte rien à l'agent.

Chaque niveau se décompose en un ou plusieurs **modules** (on note $\mathtt{M}_{\mathtt{p},\mathtt{n}}$ le nombre de modules dans le niveau $\mathsf{N}_{\mathtt{p},\mathtt{n}}$. Un module est une façon d'exécuter un niveau. L'agent ne peut exécuter qu'un seul module par niveau. À chaque module est associée une qualité **Q** et une distribution de probabilités de consommation de ressources $\mathcal{P}r$. Nous notons $\mathsf{m}_{\mathtt{p},\mathtt{n},\mathtt{m}}$ le module $\mathtt{m}$ du niveau $\mathsf{N}_{\mathtt{p},\mathtt{n}}$. La qualité $\mathbf{Q}_{\mathtt{p},\mathtt{n},\mathtt{m}}$ et la distribution de probabilités de consommation de ressources $\mathcal{P}r_{\mathtt{p},\mathtt{n},\mathtt{m}}$ sont indexées de la même façon.

La **qualité Q** est un réel positif qui représente numériquement le gain obtenu après réalisation du module. Dans le cas du robot explorateur, la qualité représente le degré de satisfaction[31] de ses contrôleurs qui attendent les résultats sur Terre.

La **distribution de probabilité** de consommation de ressources $\mathcal{P}r_{\mathtt{p},\mathtt{n},\mathtt{m}}$ de chaque module représente l'incertitude qui existe dans notre modèle. C'est une fonction $\mathcal{P}r : \mathtt{R} \rightarrow [0,1]$ qui donne la probabilité de consommer une quantité de ressources quand l'agent exécute le module $\mathsf{m}_{\mathtt{p},\mathtt{n},\mathtt{m}}$

[31] Nous nous limitons à un gain mono critère.

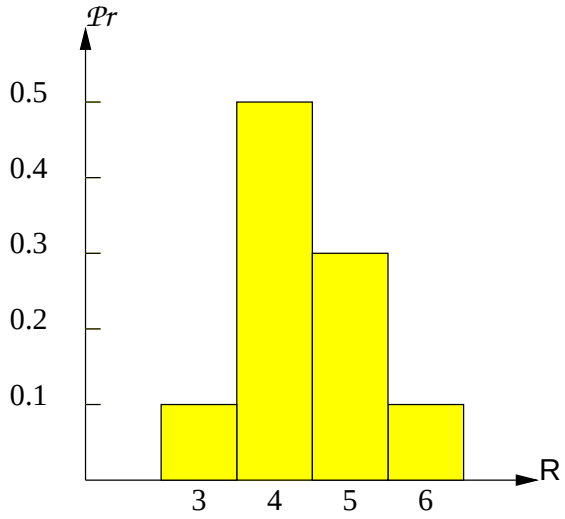


FIG. 5.3 – Distribution discrète de probabilités de consommation de ressources pour un module donné, avec un seul type de ressource.

explications *pour la figure 5.3 : cette distribution de probabilités de consommation de ressources pour un module quelconque est discrétisée. Elle nous indique par exemple que la probabilité de consommer trois unités de ressource est* 0.1. *Le domaine de cette fonction est* $[3, 6]$.

(R représente l'espace des ressources consommables).

Les **ressources consommables** sont des quantités mesurables et décroissantes. Les quantités de ces ressources consommables diminuent au fur et à mesure que la mission se poursuit, après chaque module, sans jamais augmenter pendant la mission. Parmi celles-ci on peut citer par exemple : le **temps** restant pour effectuer la mission, l'**énergie** dans les batteries ou l'**espace mémoire** restant sur le disque de données, comme le montre la figure 5.4. Nous justifions assez facilement que l'énergie dans les batteries est une ressource consommable décroissante, puisque le robot ne charge ses batteries qu'à la fin de sa mission. Le temps est aussi une ressource consommable : nous considèrons la journée comme la durée totale de la mission, l'agent dispose donc d'une quantité de temps égale à 24 heures qui va diminuer jusqu'à 0 durant la journée.

Note : dans ce chapitre, nous nous limitons à une seule ressource consommable, comme l'on proposé les auteurs de [Mouaddib et Zilberstein, 2000].

Si le robot ne termine pas une PRU la récompense[32] obtenue est nulle. Sinon, la récompense est égale à la somme des qualités **Q** obtenues dans les modules qui ont été exécutés.

Nous avons maintenant entre les mains toutes les informations nécessaires à la construction d'un modèle décisionnel de Markov pour calculer la politique qui permettra au robot d'exécuter

[32] Dans [Mouaddib et Zilberstein, 2000] les auteurs ont préféré parler d'utilité non pénalisée par la consommation de ressources **U**(**Q**) obtenue quand une PRU est terminée.

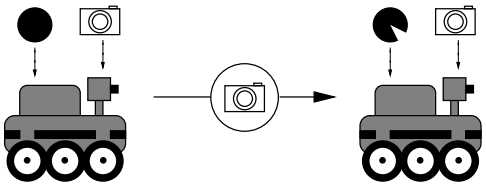


FIG. 5.4 – Consommation d'espace disque.

explications *pour la figure 5.4 : le robot prend une photo, puis l'enregistre sur son disque dur. La place existante sur le disque dur est fixe, et diminue après chaque photo prise.*

sa mission. Nous allons donc présenter le mécanisme de contrôle du raisonnement progressif.

Le mécanisme de contrôle de raisonnement progressif se fait en plusieurs étapes :

- la construction d'un MDP à partir de la mission
- le calcul d'une politique globale à partir du MDP
- l'exécution de la mission en suivant cette politique.

Le mécanisme de contrôle global prévoit un contrôle total de la mission, c'est-à-dire que la politique globale est calculée avant d'exécuter la mission et pour toute la mission. Puisque nous avons opté pour un contrôle à base de processus décisionnels de Markov, nous allons présenter chacun des éléments de ce MDP, à savoir, l'**espace d'états**, l'**espace d'actions**, la fonction de **transition**, et la fonction de **récompense**.

5.1.1 Les états de l'agent

L'état de l'agent dépend entre autre de la PRU qu'il exécute et du niveau qu'il a atteint dans celle-ci. On accumule dans l'état la qualité de tous les modules qui ont été exécutés dans cette PRU, c'est pourquoi la qualité accumulée $\mathbf{Q}$ est un paramètre de l'état. De plus, la décision à prendre par l'agent dépend des tâches à venir donc des ressources consommables restantes, et des tâches à effectuer pour le reste de la mission.

L'état de l'agent va être décrit par quatre variables : les ressources restantes $\mathtt{r}$, la qualité $\mathbf{Q}$ accumulée dans la PRU courante, l'unité de raisonnement progressif courante et son indice $\mathtt{p}$ puis l'indice $\mathtt{n}$ du dernier niveau exécuté dans cette PRU. Si par exemple, le robot vient de terminer le $2^{\text{ème}}$ niveau de la PRU numéro 5, que les modules précédemment exécutés lui ont rapporté une qualité de 15, et qu'il lui reste 153 unités de temps, l'état sera noté $\mathbf{s} = [153, 15, 2, 5]$. Il peut arriver que l'agent commence une tâche qui va consommer plus de ressources qu'il n'en avait au départ, on regroupe donc tous les états dont la ressource restante serait négative dans un état d'échec $\mathbf{s}_{\mathtt{echec}}$.

Formellement, on écrira que : $\mathcal{S} = \{[\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}], \mathtt{r} \in \mathtt{R}\} \cup \{\mathbf{s}_{\mathtt{echec}}\}$.

De plus, on crée un ensemble d'états pour un niveau 0 fictif qui correspond pour chaque PRU

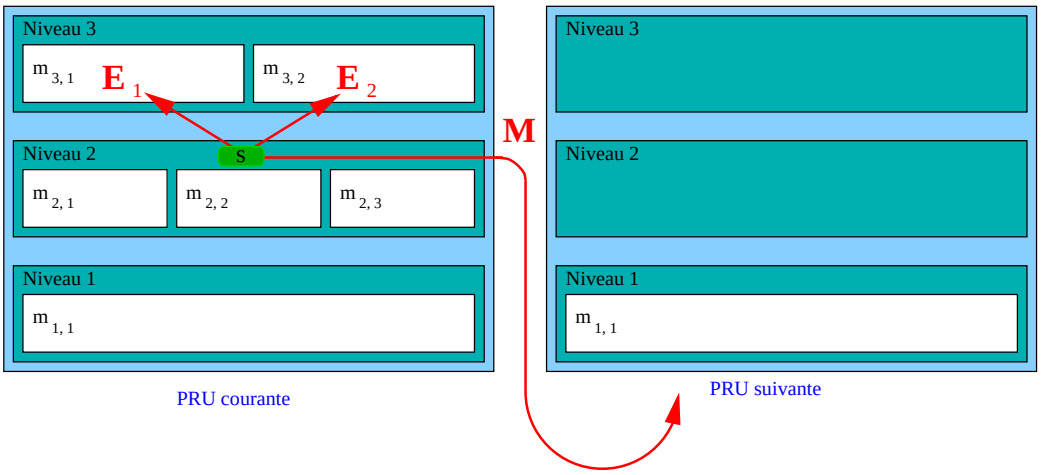


FIG. 5.5 – L'espace d'actions

explications *pour la figure 5.5 : l'agent a terminé le niveau 2 de la PRU courante. Deux types d'actions peuvent être entreprises : exécuter un des modules du niveau suivant,* $\mathbf{E}_1$ *ou* $\mathbf{E}_2$*, ou alors abandonner la PRU courante, et se placer au début d'une PRU suivante.*

aux états d'entrée de cette PRU. Dans cet ensemble d'états, la qualité est toujours nulle, puisque l'agent n'a encore exécuté aucun module de cette PRU.

5.1.2 Les actions de l'agent

Seulement deux types d'actions sont possibles (voir figure 5.5), :

- l'action $\mathbf{E}_{\mathsf{m}}$ qui permet d'exécuter un des modules m du niveau suivant de la PRU courante.
- l'action **M** qui permet de changer de PRU

L'agent va donc avoir après chaque niveau le choix entre l'amélioration de la tâche courante (la PRU courante) ou l'abandon de la PRU pour la suivante. Originellement une action **S** permettait de sauter un niveau, c'est-à-dire de le passer sans exécuter aucun module. Nous avons supprimé l'action **S** et rajouté un module vide dans les niveaux qui pouvaient être sautés. Ce module produit une qualité nulle, et ne consomme pas de ressource. Exécuter un module skip est donc équivalent à une action skip.

Formellement, on écrira que : $\mathcal{A} = \{\mathbf{E}_{\mathtt{m}}, \mathbf{M}\}$

Quand l'agent choisit d'exécuter un niveau, il doit choisir parmi un des modules disponibles de ce niveau. Les actions **E** sont indexées par le module choisi. Si le dernier niveau est atteint, et que plus aucun niveau n'est disponible, c'est-à-dire que la tâche est achevée, alors l'action **E** n'est plus disponible, seule l'action **M** est possible.

Si l'agent décide d'arrêter d'améliorer la PRU courante, il choisit l'action $\mathbf{M}$. Il va alors se trouver immédiatement au début[33] de la PRU suivante, ou il pourra choisir, une fois de plus, quel module du premier niveau de cette PRU il souhaite exécuter, ou tout simplement réappliquer l'action $\mathbf{M}$, c'est-à-dire ne rien faire du tout sur cette tâche, l'ignorer, et passer à la suivante.

Par exemple, le robot peut avoir repéré une roche, mais se rendre compte que s'il la ramasse, il va perdre un temps précieux pour plus tard. Il peut donc décider d'arrêter la tâche courante, c'est-à-dire ne pas ramasser cette roche, et passer directement à un autre site, pour repérer une autre roche, ou encore continuer vers un deuxième site sans rien faire sur le premier. Le robot ne pourra par contre jamais revenir sur ses pas pour reprendre un site abandonné.

5.1.3 La fonction de transition

La fonction de transition que nous notons $\mathcal{P}r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0,1]$ modélise l'incertitude de notre système.

L'action $\mathbf{M}$ est déterministe. Si l'agent choisit de changer d'unité de raisonnement progressif, il garde toutes ses ressources restantes. Ainsi $\mathcal{P}r([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}], \mathbf{M}, [\mathtt{r}, 0, \mathtt{p}+1, 0]) = 1$. La qualité accumulée dans la nouvelle PRU devient nulle. Le deuxième zéro représente le fait que l'agent est placé juste avant le premier niveau de la $\mathsf{PRU}_{\mathtt{p}+1}$.

L'action $\mathbf{E}$ est probabiliste pour tenir compte de l'incertitude quant à la consommation de ressources. La distribution de probabilités est décrite dans le module à exécuter (celle de la figure 5.3). Ainsi $\mathcal{P}r([\mathtt{r}, \mathtt{p}, \mathbf{Q}, \mathtt{n}], \mathbf{E}_{\mathtt{m}}, [\mathtt{r} - \Delta\mathtt{r}, \mathbf{Q} + \mathbf{Q}_{\mathtt{p},\mathtt{n},\mathtt{m}}, \mathtt{p}, \mathtt{n}+1]) = \mathcal{P}r(\Delta\mathtt{r} | \mathsf{m}_{\mathtt{p},\mathtt{n},\mathtt{m}})$.

5.1.4 La fonction de récompense

Dans notre application robotique, le mécanisme de contrôle du robot, tel qu'il a été pensé dans [Mouaddib et Zilberstein, 2000] prévoit de récompenser le robot après chaque PRU pour ce qu'il a fait. Aucune pénalité ne lui est infligée pour les ressources consommées. La qualité est une variable de l'état : ceci permet de cumuler la qualité niveau après niveau (en fonction des modules effectués) et de ne récompenser le robot que si la PRU a été entièrement effectuée à la fin de celle-ci.

$$\mathcal{R}([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{N_p}]) = \mathbf{Q} \tag{5.1}$$

$$\mathcal{R}([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n} < \mathtt{N_p}]) = 0 \tag{5.2}$$

Même si on ne pénalise pas directement le robot pour avoir consommé des ressources dans une PRU donnée, la consommation locale entraîne une perte possible de valeur pour le futur, puisque ce qui est dépensé maintenant ne sera plus disponible plus tard dans la mission. Nous

[33] C'est-à-dire à la fin d'un niveau 0 fictif.

verrons dans la section suivante comment, grâce au coût occasionné, nous pouvons tenir compte de cette pénalité future.

5.1.5 La fonction de valeur

Afin de pouvoir calculer une politique, il nous faut une fonction de valeur. La valeur en sortie d'une PRU ne dépend que de la qualité qui y a été accumulée, et du temps restant après l'obtention du résultat. Pour contrôler la $\mathsf{PRU}_{\mathtt{p}}$, il suffit de connaître la fonction de valeur liée à l'action d'exécution des modules $\mathbf{E}$. La valeur d'un état au dernier niveau est la récompense pour avoir obtenu ce résultat, donc :

$$V([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{N_p}]) \quad = \quad \mathcal{R}([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{N_p}]) \tag{5.3}$$

La valeur d'un état avant le dernier niveau dépend de la valeur espérée en exécutant les niveaux supérieurs, et en choisissant le meilleur module :

$$V_{\mathbf{E}}([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]) \quad = \quad \max_{\mathtt{m}} EV(\mathbf{E}_{\mathtt{m}}, [\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]) \tag{5.4}$$

où EV est une fonction de valeur espérée qui s'écrit de la façon suivante :

$$EV(\mathbf{E}_{\mathtt{m}}, [\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]) = \underbrace{\sum_{\mathbf{Q}', \Delta\mathtt{r} > \mathtt{r}} \mathcal{P}r_{\mathtt{p},\mathtt{n}+1,\mathtt{m}}((\mathbf{Q}', \Delta\mathtt{r}) | \, \mathbf{Q}) \times V(\mathbf{s}_{\mathtt{echec}})}_{\text{Si les ressources sont épuisées}} + \underbrace{\sum_{\mathbf{Q}', \Delta\mathtt{r} \leq \mathtt{r}} \mathcal{P}r_{\mathtt{p},\mathtt{n}+1,\mathtt{m}}((\mathbf{Q}', \Delta\mathtt{r}) | \, \mathbf{Q}) \times V([\mathtt{r} - \Delta\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}+1])}_{\text{Si le module est exécuté}}$$

L'équation 5.3 permet de donner une valeur aux états du dernier niveau de la PRU, tandis que l'équation 5.4 évalue les états du premier à l'avant dernier niveau. Quand il arrive au dernier niveau, l'agent obtient une récompense. Sinon, dans les niveaux inférieurs, on calcule ce que l'on peut espérer gagner en exécutant chaque module (avec la fonction EV). On compare ensuite les valeurs obtenues pour chaque module, et on ne garde que le maximum, comme pour une équation de Bellman. $V(\mathbf{s}_{\mathtt{echec}})$ est le gain obtenu quand la ressource est épuisée, il est nul.

Rappelons que la fonction de valeur est une mesure de l'espérance de gain. Si lors de l'exécution l'agent n'exécute pas le dernier niveau de la PRU avec ce système d'équations, il ne gagne rien en retour. Les récompenses ne sont pas directement additives, elles sont cumulées tout au long de l'exécution de la PRU puis obtenues tout à la fin.

Pour contrôler plusieurs PRUs, il suffit de rajouter l'équation suivante, liée à la valeur pour l'action changement de PRU $\mathbf{M}$:

$$V_{\mathbf{M}}([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{N_p}]) = \mathcal{R}([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{N_p}]) + V([\mathtt{r}, \mathbf{0}, \mathtt{p}+1, 0]) \quad (5.5)$$

$$V_{\mathbf{M}}([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n} < \mathtt{N_p}]) = \underbrace{\mathcal{R}([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n} < \mathtt{N_p}])}_{0} + V([\mathtt{r}, \mathbf{0}, \mathtt{p}+1, 0]) \quad (5.6)$$

$$V_{\mathbf{M}}([\mathtt{r}, \mathbf{Q}, \mathbf{P}{-}1, \mathtt{N_{P-1}}]) = \mathcal{R}_{\mathbf{P}}([\mathtt{r}, \mathbf{Q}, \mathbf{P}{-}1, \mathtt{N_{P-1}}]) + 0 \quad (5.7)$$

$$V_{\mathbf{M}}([\mathtt{r}, \mathbf{Q}, \mathbf{P}{-}1, \mathtt{n} < \mathtt{N_{P-1}}]) = \underbrace{\mathcal{R}([\mathtt{r}, \mathbf{Q}, \mathbf{P}{-}1, \mathtt{n} < \mathtt{N_{P-1}}])}_{0} + 0 = 0 \quad (5.8)$$

La valeur liée à un changement de PRU est celle obtenue directement en commençant la PRU suivante depuis la base, un niveau 0 fictif qui montre que l'exécution de cette PRU n'est pas commencée. La nouvelle qualité est nulle. Pour la dernière PRU, l'action **M** ne rapporte rien, l'agent obtient une récompense s'il termine cette PRU (équation 5.7), sinon rien (équation 5.8), **P** désigne le nombre de PRUs dans la mission, $\mathbf{P}{-}1$ est donc l'indice de la dernière PRU de la mission. Pour les autres PRUs la valeur est égale à la récompense + ce qui peut être gagné plus tard si on finit la PRU (équation 5.5) sinon la valeur est égale à ce qui peut être gagné plus tard (équation 5.6).

Finalement, en gardant la meilleure action, on retrouve l'équation de Bellman, adaptée au contexte du raisonnement progressif.

$$V([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]) = \max(V_{\mathbf{M}}, V_{\mathbf{E}}) \quad (5.9)$$

$$\pi([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]) = \underset{\mathbf{E},\mathbf{M}}{\operatorname{argmax}}(V_{\mathbf{M}}, V_{\mathbf{E}}) \quad (5.10)$$

Le modèle de Markov qui découle du raisonnement progressif a des propriétés particulières que nous allons énoncer afin de pouvoir en tirer profit lors de la résolution du problème, c'est-à-dire lors du calcul de la fonction de valeur. Un dénombrement des états va nous permettre d'étudier la complexité du problème.

5.1.6 Étude de complexité

Espaces bornés

On remarquera que plusieurs espaces sont bornés :

– pendant la mission, l'espace des **ressources** R est borné, puisque la quantité de ressources allouées au départ $\mathtt{r}_D$ ne peut que diminuer pendant la mission, et ne peut passer en dessous de zéro donc :

$$\forall \mathtt{r} \in \mathtt{R}, \qquad 0 \leq \mathtt{r} \leq \mathtt{r}_D \quad (5.11)$$

– au sein même d'une unité de raisonnement progressif, le nombre de ressources $\mathtt{r}_{cons}$ qui pourront être consommées est borné. On peut ne rien consommer du tout, ou dans le cas extrême, consommer un maximum de ressources qui est la somme de ce que peut consommer au maximum[34] le module le plus gourmand en ressources de chaque niveau (en supposant que tous les niveaux sont exécutés).

$$0 \leq \quad \mathtt{r} \quad \leq \sum_{\mathtt{n}=1}^{\mathtt{N_p}} \max_{\mathtt{m} \in \mathtt{M_n}} \left(\max_{\mathtt{r}} \{ \mathtt{r} \in Dom(\mathcal{P}r_{\mathtt{p,n,m}}) \} \right) = \mathtt{r}_{\mathtt{p}}^{max} \tag{5.12}$$

– au sein d'une même unité de raisonnement progressif, la qualité totale qui sera obtenue en sortie est comprise entre zéro si on ne fait rien, et la somme de toutes les qualités des modules dont la qualité est la plus grande à chaque niveau, soit :

$$0 \leq \quad \mathbf{Q}_{\mathtt{p}}^{\max} \quad \leq \sum_{\mathtt{n}=1}^{\mathtt{N_p}} \max_{\mathtt{m} \in \mathtt{M_n}} \mathbf{Q}_{\mathtt{p,n,m}} \tag{5.13}$$

– de ce fait, la qualité totale de la mission est également bornée, et la fonction de valeur V aussi.

L'espace d'états $\mathcal{S}$

Un état est composé de :
– du montant $\mathtt{r}$ de ressources restantes,
– de l'indice de la PRU courante $\mathtt{p} \in [1, ..., \mathbf{P}]$,
– de l'indice du dernier niveau exécuté $\mathtt{n} \in [0, ..., \mathtt{N}]$,
– de la qualité accumulée $\mathbf{Q}$.

L'espace d'états $\mathcal{S}$ pour la mission est fini et chacun de ses paramètres est borné. On peut calculer facilement $|\mathcal{S}|$ en faisant la somme du nombre d'états possibles dans chaque unité de raisonnement progressif. Dans une PRU, nous allons regrouper tous les états qui ont la même qualité accumulée à un niveau donné et noter cet ensemble $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$. Une fois que nous aurons calculé le nombre de $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$ dans chaque PRU, il ne restera plus qu'à calculer le nombre de valeurs que peut prendre la ressource sur chaque sous-espace d'états.

Dans une PRU $\mathtt{p}$, il y a $\mathtt{N_p}$ niveaux. La qualité $\mathbf{Q}$ accumulée à un niveau $\mathtt{n}$ dépend des différentes qualités accumulées dans le niveau précédent. Il y a au plus autant de qualités accumulées possibles à un niveau que le produit de toutes les qualités accumulées possibles du niveau précédent et des qualités des différent modules du niveau courant. Pour un niveau $\mathtt{n}$ donné le nombre de qualités accumulées possibles $|\mathbf{Q}_{\mathtt{n}}| = |\mathtt{M}_{\mathtt{p,n}}| \times |\mathbf{Q}_{\mathtt{n}-1}|$, $|\mathtt{M}_{\mathtt{p,n}}|$ étant le nombre de modules du niveau concerné. Donc dans une PRU $\mathtt{n}$, nous avons un nombre de qualité accumulées

[34] Nous considérons que cette quantité est effectivement bornée dans les distributions de probabilité de consommation de ressources.

possibles inférieur ou égal à $\sum_{\mathtt{n}=1}^{\mathtt{N_p}} |\mathbf{Q}_{\mathtt{p,n}}|$. Il faut rajouter à cela le sous-espace d'états du niveau 0 correspondant aux états entrants de chaque PRU.

$$|\mathbf{Q}_{\mathtt{p}}| \leq 1 + \sum_{\mathtt{n}=1}^{\mathtt{N_p}} \prod_{\mathtt{n}'=1}^{\mathtt{n}} \mathtt{M}_{\mathtt{p,n}'} \tag{5.14}$$

Maintenant, dénombrons le nombre de ressources restantes possibles à un niveau $\mathtt{n}$ donné d'une PRU $\mathtt{p}$ donnée, à $\mathbf{Q}$ fixée. La quantité de ressources maximale qui peut être dépensée dans une PRU $\mathtt{p}$, et qui garantit[35] l'obtention d'une qualité accumulée maximale est notée $\mathtt{r}_{\mathtt{p}}^{max}$ et est égale à la partie droite de l'inéquation 5.12.

Pour la mission dont les PRUs sont indicées par $\mathtt{p} \in [1, ..., \mathbf{P}]$, on sait que le nombre total de ressources consommables au maximum dans la dernière PRU $\mathbf{P}$ est $\mathtt{r}_{\mathbf{P}}^{max}$ et à partir de l'avant dernière $\mathsf{PRU}_{\mathbf{P}-1}$ est égal à la somme des deux : $\mathtt{r}_{\mathbf{P}}^{max} + \mathtt{r}_{\mathbf{P}-1}^{max}$. Finalement le nombre total de ressources consommables au maximum dans les PRUs de $\mathtt{p}$ à $\mathbf{P}$ est la somme de toutes les ressources maximum consommables dans les PRUs considérées. Donc, le nombre d'états possibles dans un sous-espace $\mathcal{S}_{\mathbf{Q},\mathtt{p,n}}$ est :

$$|\mathcal{S}_{\mathbf{Q},\mathtt{p,n}}| \leq \sum_{\mathtt{p}'=\mathtt{p}}^{\mathbf{P}} \mathtt{r}_{\mathtt{p}'}^{max} \tag{5.15}$$

L'agent peut commencer la mission avec $\mathtt{r}_{\mathsf{total}}^{max} = \sum_{\mathtt{p}=1}^{\mathbf{P}} \mathtt{r}_{\mathtt{p}}^{max}$ et ceci oblige, dans le pire des cas, à générer $\mathtt{r}_{\mathsf{total}}^{max}$ états par sous-espace d'états $\mathcal{S}_{\mathbf{Q},\mathtt{p,n}}$.

Le nombre total d'états possibles dans la mission est donc dans le pire des cas pour une PRU donnée égal à $|\mathcal{S}_{\mathbf{Q},\mathtt{p,n}}|.|\mathbf{Q}_{\mathtt{p}}|$ (voir figure 5.6) . Le nombre total d'états dans la mission est donc inférieur ou égal à :

$$|\mathcal{S}| \leq \min\left(\mathtt{r}_{\mathsf{total}}^{max}, \mathtt{r}_D\right) \times \left(\sum_{\mathtt{p}=1}^{\mathbf{P}} |\mathbf{Q}_{\mathtt{p}}|\right) \tag{5.16}$$

$$|\mathcal{S}| \leq \min\left(\sum_{\mathtt{p}=1}^{\mathbf{P}} \mathtt{r}_{\mathtt{p}}^{max}, \mathtt{r}_D\right) \times \left(\sum_{\mathtt{p}=1}^{\mathbf{P}} \left(1 + \sum_{\mathtt{n}=1}^{\mathtt{N_p}} \prod_{\mathtt{n}'=1}^{\mathtt{n}} \mathtt{M}_{\mathtt{p,n}'}\right)\right) \tag{5.17}$$

Cependant, cette mesure reste quelque peu pessimiste. Nous pouvons d'ores et déjà réduire la taille de l'espace d'états PRU par PRU. En effet, à l'exécution de la dernière $\mathsf{PRU}_{\mathbf{P}}$, s'il reste $\mathtt{r}_{\mathbf{P}}^{max}$ ressources à l'agent, celui-ci pourra obtenir une valeur maximale en exécutant tous les meilleurs modules dans tous les niveaux de cette $\mathsf{PRU}_{\mathbf{P}}$. Si l'agent disposait de plus de $\mathtt{r}_{\mathbf{P}}^{max}$ pour exécuter la dernière PRU, il obtiendrait toujours cette valeur maximale. Nous allons donc considérer que pour la dernière PRU tous les états $[\mathtt{r}, \mathbf{Q}, \mathbf{P}, \mathtt{n}]$ où $\mathtt{r} > \mathtt{r}_{\mathbf{P}}^{max}$ sont agrégés dans le même état $[\mathtt{r}_{\mathbf{P}}^{max}, \mathbf{Q}, \mathbf{P}, \mathtt{n}]$. De ce fait, chaque sous-espace d'états $\mathcal{S}_{\mathbf{Q},\mathbf{P},\mathtt{n}}$ sera de taille $\mathtt{r}_{\mathbf{P}}^{max}$. Nous

[35] S'il y a plus de ressource que $\mathtt{r}_{\mathtt{p}}^{max}$, la qualité accumulée sera elle aussi maximale.

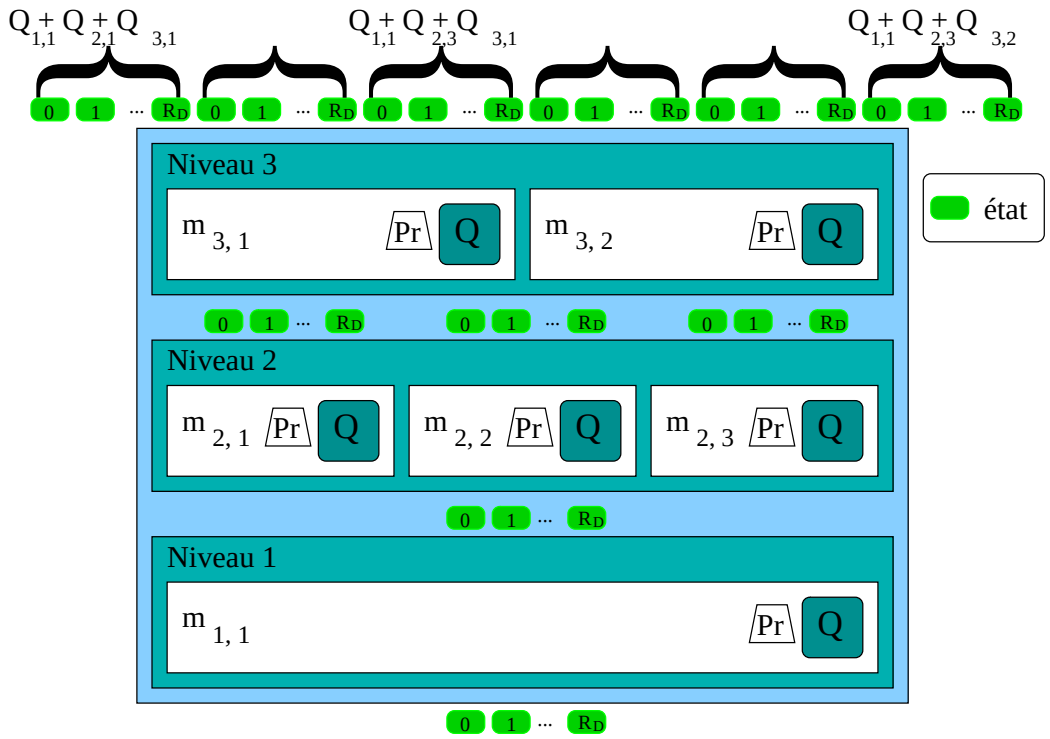


FIG. 5.6 – L'espace d'états pour une seule PRU

explications *pour la figure 5.6 : chaque état* $[\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]$ *de la* $\mathsf{PRU}_{\mathtt{p}}$ *est représenté par un rectangle dans lequel est inscrit le nombre de ressources restantes (de* 0 *à* $\mathtt{r}_D$*). On a noté le nombre de ressources restantes à l'intérieur de chaque état. Les qualités accumulées se multiplient : il y a trois valeurs de qualités accumulées avant le niveau* 3 *puis six après ce niveau. Le nombre de modules par niveau joue donc un rôle essentiel dans la taille de l'espace d'états. Les regroupements d'états, avec* $\mathtt{p}$, $\mathtt{n}$ *et* $\mathbf{Q}$ *fixés représentent les sous-espaces d'états* $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$

allons ainsi réduire l'espace d'états par récurrence. En ce qui concerne l'avant dernière unité de raisonnement progressif, $\mathsf{PRU}_{\mathbf{P}-1}$, avoir une quantité $\mathtt{r}_{\mathbf{P}}^{max} + \mathtt{r}_{\mathbf{P}-1}^{max}$ de ressources permet à l'agent d'obtenir à coup sûr un gain maximal, et donc nous limitons la taille de chaque sous-espace d'états $\mathcal{S}_{\mathbf{Q},\mathbf{P}-1,\mathtt{n}}$ de cette unité de raisonnement progressif à $\mathtt{r}_{\mathbf{P}}^{max} + \mathtt{r}_{\mathbf{P}-1}^{max}$. Nous pouvons montrer par récurrence que l'espace d'états total pour la mission peut se limiter à :

$$|\mathcal{S}| \leq \sum_{\mathtt{p}=1}^{\mathbf{P}} \left(\min \left(\sum_{\mathtt{p}'=\mathtt{p}}^{\mathbf{P}} \mathtt{r}_{\mathtt{p}'}^{max}, \mathtt{r}_D \right) \right) \times \left(1 + \sum_{\mathtt{n}=1}^{\mathtt{N}_{\mathtt{p}}} \prod_{\mathtt{n}'=1}^{\mathtt{n}} \mathtt{M}_{\mathtt{p},\mathtt{n}'} \right) \tag{5.18}$$

En supposant que toutes les PRUs de la mission sont identiques (donc toutes du même type que la dernière), l'équation se simplifie en :

$$|\mathcal{S}| \leq \sum_{\mathtt{p}=1}^{\mathbf{P}} \left(\min \left(\sum_{\mathtt{p}'=\mathtt{p}}^{\mathbf{P}} \mathtt{r}_{\mathbf{P}}^{max}, \mathtt{r}_D \right) \right) \times \underbrace{\left(1 + \sum_{\mathtt{n}=1}^{\mathtt{N_p}} \prod_{\mathtt{n}'=1}^{\mathtt{n}} \mathtt{M}_{\mathtt{p},\mathtt{n}'} \right)}_{\mathbf{C}}$$

Nous supposons que l'agent peut disposer du montant maximum de ressource et supprimons ainsi $\mathtt{r}_D$ de l'équation pour le pire cas ; $\mathbf{C}$ est une constante qui représente le nombre de sous-espaces d'états $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$ dans chaque PRU :

$$|\mathcal{S}| \leq \sum_{\mathtt{p}=1}^{\mathbf{P}} \left(\sum_{\mathtt{p}'=\mathtt{p}}^{\mathbf{P}} \mathtt{r}_{\mathbf{P}}^{max} \right) . \mathbf{C} \tag{5.19}$$

$$\leq \mathbf{C} . \sum_{\mathtt{p}=1}^{\mathbf{P}} (\mathbf{P} - \mathtt{p}) . \mathtt{r}_{\mathbf{P}}^{max} \tag{5.20}$$

$$\leq \mathbf{C} . \mathtt{r}_{\mathbf{P}}^{max} . \frac{\mathbf{P} . (\mathbf{P} - 1)}{2} \tag{5.21}$$

Cette simplification nous permet de montrer que la taille de l'espace d'états augmente donc au carré avec le nombre de PRU, et linéairement avec la quantité de ressources que l'agent peut consommer dans chaque PRU.

Nous avons présenté le MDP qui modélise la mission, et le problème principal de celui-ci est la taille de l'espace d'états. Nous allons maintenant montrer comment résoudre le problème de décision markovien associé, pour obtenir avant l'exécution une politique optimale d'exécution.

5.2 Deux algorithmes pour calculer la politique de contrôle globale

Le mécanisme de contrôle global se base sur un calcul *a priori* de la politique pour l'ensemble de la mission.

5.2.1 Résolution par programmation dynamique

Nous voyons clairement avec le système d'équations qui découle de l'équation (5.9) du paragraphe précédent que la valeur d'un état ne dépend que de la valeur des états des niveaux suivants et des PRU suivantes. Il n'y aura donc nullement besoin d'appliquer un algorithme du type de value iteration ou policy iteration vus dans le chapitre 2. Ce MDP est sans cycle et à horizon fini. Les états terminaux sont identifiables : ce sont les états du dernier niveau de la dernière PRU et l'état d'échec. Une fois ces états évalués, il suffit de remonter petit à petit pour

trouver la valeur de chaque état du MDP, et de leur associer la meilleure action. Nous allons donc développer un algorithme de programmation dynamique avec une mémoization[36].

Pour calculer la politique il va donc falloir :

1. **créer** chaque état (récursivement), grâce au chaînage avant,
2. **évaluer** ces états par chaînage arrière, et
3. **associer** à chacun d'eux une action, afin d'obtenir une politique.

L'algorithme 5 d' **exploration de l'espace d'états** est un algorithme de **programmation dynamique**. Il va chercher à créer les états qui ne sont pas encore dans $\mathcal{S}$, les garder en mémoire dans une liste associative ou un dictionnaire (qui est $\mathcal{S}$ justement), pour garder en mémoire leur valeur et ne pas les réévaluer (principe de mémoization). Cette récursion se termine bien puisque soit l'agent n'a plus de ressources, soit il a exécuté le dernier niveau de la dernière PRU.

Pour la dernière PRU, quand l'algorithme arrive sur un état terminal, celui-ci est évalué, puis tous ses prédécesseurs le sont, en remontant le graphe d'états par chaînage arrière (ceci se fait naturellement quand on applique l'action **E**). On obtient finalement une politique optimale et exhaustive sur l'ensemble des états qui peuvent être atteints depuis l'état initial.

Cet algorithme nous permet d'obtenir une politique optimale pour le contrôle de la mission. L'avantage est que notre structure de mission est acyclique et que par conséquent chaque état n'est évalué qu'une seule fois. Le temps de calcul est proportionnel au nombre d'états du MDP. Un inconvénient majeur de cette méthode : l'espace d'états doit être pratiquement entièrement généré et stocké dans une structure adéquate (un dictionnaire ou liste d'association). La complexité de cet algorithme est proportionnelle à la taille de l'espace d'états, c'est-à-dire la dimension de l'espace des ressources multiplié par le nombre de PRUs au carré.

5.2.2 L'algorithme d'énumération d'états par niveaux

Le deuxième algorithme est aussi de la programmation dynamique. Mais, au lieu d'explorer l'espace d'états et de le construire au fur et à mesure, nous allons générer tous les états possibles de la mission PRU par PRU puis niveau par niveau. On génère donc l'espace d'états tout entier, puis on l'évalue, en commençant par les états terminaux. Nous avons constaté que lors de la génération de l'espace d'états par exploration (comme dans la section précédente), la quasi totalité de l'espace d'états était générée. Nous avons choisi de générer cet espace d'états dans un certain *ordre*. Cet ordre nous permettra de faire des économies de calcul et de représentation pour le raisonnement progressif multi-ressources que nous présentons dans le chapitre suivant.

Pour générer l'espace d'états de façon exhaustive, nous le découpons en sous-espaces d'états : à chaque PRU `p` correspond un sous-espace d'états indicé par `p`. Puis dans une PRU donnée,

[36] La memoization consiste à enregistrer les états déjà rencontrés et leur valeur de façon à ne pas calculer deux fois la valeur d'un état connu.

Données : une mission $[\mathsf{PRU}_1, \ldots, \mathsf{PRU}_{\mathbf{P}}]$
Résultat : π, *valeur*
$\mathcal{S} = \emptyset$
evalueEtat $([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}])$: fonction récursive
si $\mathtt{r} < 0$ *cas de l'échec* **alors**
  retouner 0
finsi
si $([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}] \in \mathcal{S})$ *pour ne pas évaluer deux fois un état* **alors**
  retouner valeur$([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}])$
finsi
$\mathcal{S}$.**ajouter**$([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}])$
si $\mathtt{n} == \mathtt{N_p}$ **alors**
  si $\mathtt{p} == \mathbf{P}$ **alors**
    $valeur = \mathcal{R}_{\mathtt{p}}([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{N_p}])$
  sinon
    $valeur = \mathcal{R}_{\mathtt{p}}([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{N_p}]) + \mathbf{evalueEtat}([\mathtt{r}, 0, \mathtt{p}+1, 0])$
    $\pi([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]) = \mathbf{M}$
  finsi
sinon
  $valeur_{\mathbf{E}} =$
  $\max\limits_{1 \le \mathtt{m} \le \mathtt{M}_{\mathtt{p},\mathtt{n}+1}} \left(\sum\limits_{\Delta\mathtt{r} \le \mathtt{r}} \mathcal{P}r(\Delta\mathtt{r} | \mathtt{m}_{\mathtt{p},\mathtt{n}+1,\mathtt{m}}) \times \left(\mathbf{evalueEtat}([\mathtt{r} - \Delta\mathtt{r}, \mathbf{Q} + \mathbf{Q}_{\mathtt{p},\mathtt{n}+1,\mathtt{m}}, \mathtt{p}, \mathtt{n}+1]) \right) \right)$
  si $\mathtt{p} == \mathbf{P}$ **alors**
    $valeur_{\mathbf{M}} = 0$
  sinon
    $valeur_{\mathbf{M}} = \mathbf{evalueEtat}([\mathtt{r}, 0, \mathtt{p}+1, 0])$
  finsi
  $\pi([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]) = \underset{\mathbf{E},\mathbf{M}}{\operatorname{argmax}}(valeur_{\mathbf{E}}, valeur_{\mathbf{M}})$
  $valeur = \max\limits_{\mathbf{E},\mathbf{M}}(valeur_{\mathbf{E}}, valeur_{\mathbf{M}})$
finsi
$\mathcal{S}([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{N_p}])$.**valeur** $= valeur$
retourner *valeur*

Algorithme 5 : L'algorithme d'évaluation par programmation dynamique

on regroupe les états par niveau. Dans chaque niveau, il existe plusieurs qualités accumulées

différentes. On regroupe les états par qualité accumulée, et on crée toutes les clés[37] $\mathbf{Q}_{\mathtt{p},\mathtt{n}}$ des sous-ensembles $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}} = \{\mathbf{s} = [\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}], \mathtt{r} \in \mathtt{R}\}$. Ces sous-ensembles sont générés récursivement dans une PRU donnée par l'algorithme récursif 6 qui trouve par exploration l'ensemble des qualités accumulées possibles de chaque niveau, c'est-à-dire pour $\mathtt{p}$ donné, il trouve tout les couples $(\mathtt{n}, \mathbf{Q})$ tel que $\mathbf{s} = [\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]$ est un état possible de la mission.

```
   Données : Clefs = ∅
 1 genereNiveau() :
 2 début
 3     pour p ∈ [0..P] faire
 4         recursifGenereNiveau(p, 0, 0)
 5     finpour
 6 fin
 7 recursifGenereNiveau(p, n, Q) :
 8 début
 9     Clefs[(p, n)].ajoute(Q)
10     si n < N_p alors
11         pour m ∈ [0..M_{p,n}] faire
12             recursifGenereNiveau(p, n+1, Q + Q_{m,p,n})
13         finpour
14     finsi
15 fin
```

Algorithme 6 : Génèrer les clefs $\mathbf{Q}_{\mathtt{p},\mathtt{n}}$ d'une mission

Une fois les clefs $\mathbf{Q}_{\mathtt{p},\mathtt{n}}$ des $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$ trouvées on génère les sous-ensembles d'états correspondants, et on évalue chaque état ainsi créé avec l'algorithme 7, en commençant par ceux qui se situent au dernier niveau de la dernière PRU. La première évaluation se fait grâce à la fonction de récompense. Nous générons ensuite les états de l'avant dernier niveau de la dernière PRU qui sont évalués grâce à l'équation de Bellman et aux valeurs du niveau juste au-dessus, qui vient justement d'être évalué. On continue ainsi jusqu'à ce que tous les états de la dernière PRU soient évalués. Finalement, on englobe ce processus dans une fonction itérative qui génère et calcule les états de tous les niveaux de toutes les PRUs en commençant par la dernière PRU, en finissant par la première. De cette façon, chaque état possible de la mission est évalué une seule fois. La complexité de cet algorithme est donc proportionnel au nombre total d'états possibles de la mission.

Note 1 : nous générons chaque fois les sous-espaces $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$ sur $\mathtt{R}$, pour être sûrs de prendre en

[37] Attention, on ne génère que les clefs, pas les sous-ensembles à ce stade du processus.

Données : $\mathbf{Q}, \mathtt{p}, \mathtt{n}, \mathcal{S}$

evalueNiveau$(\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}})$:

$\mathtt{r}_{max} = \min(\sum_{\mathtt{p}=\mathtt{p}'}^{\mathtt{P}} \mathtt{r}_{\mathtt{p}}^{max}, \mathtt{r}_{\mathtt{D}})$

pour $\mathtt{r} \in [0..\mathtt{r}_{max}]$ **faire**

  si $\mathtt{n} == \mathtt{N}_{\mathtt{p}}$ **alors**

    si $\mathtt{p} == \mathbf{P}$ **alors**

      $\mathcal{S}[\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}].\text{valeur} = \mathcal{R}([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{N}_{\mathtt{p}}])$

    sinon

      $\mathcal{S}[\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}].\text{valeur} = \mathcal{R}([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{N}_{\mathtt{p}}]) + \mathcal{S}([\mathtt{r}, 0, \mathtt{p}+1, 0]).\text{valeur}$

    finsi

  sinon

    $v_{\mathbf{E}} = \max_{1 \leq \mathtt{m} \leq \mathtt{M}_{\mathtt{p},\mathtt{n}+1}} \left(\sum_{\Delta\mathtt{r} \leq \mathtt{r}} \mathcal{P}r(\Delta\mathtt{r} | \mathtt{m}_{\mathtt{p},\mathtt{n}+1,\mathtt{m}}) \times \mathcal{S}[\mathtt{r} - \Delta\mathtt{r}, \mathbf{Q} + \mathbf{Q}_{\mathtt{p},\mathtt{n}+1,\mathtt{m}}, \mathtt{p}, \mathtt{n}+1].\text{valeur} \right)$

    si $\mathtt{p} == \mathbf{P}$ **alors**

      $v_{\mathbf{M}} = 0$

    sinon

      $v_{\mathbf{M}} = \mathbf{valeur}([\mathtt{r}, 0, \mathtt{p}+1, 0])$

    finsi

    $v = \max_{\mathbf{E},\mathbf{M}}(v_{\mathbf{E}}, v_{\mathbf{M}})$

    $\mathcal{S}[\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{n}].\text{valeur} = v$

  finsi

finpour

Algorithme 7 : Évaluation des états d'un niveau

compte tous les états possibles de n'importe quelle mission. Évidemment, pour une mission donnée avec $\mathtt{r}_{\mathtt{D}}$ ressources initiales, on appliquerait ce même algorithme avec un espace de ressources se limitant de 0 à $\mathtt{r}_{\mathtt{D}}$.

Note 2 : dans la dernière $\mathsf{PRU}_{\mathsf{P}}$, l'espace d'états à générer pour chaque $\mathcal{S}_{\mathsf{P},\mathtt{n},\mathbf{Q}}$ est de taille $\mathtt{r}_{\mathsf{P}}^{max}$[38]. Les valeurs des états comportant plus de ressources ne seront pas générés (même s'il sont possibles), leur valeur est égale à $V([\mathsf{P}, \mathtt{n}, \mathbf{Q}, \mathtt{r}_{\mathsf{P}}^{max}])$ puisque les ressources disponibles supplémentaires constituent un excédent inutile. En partant de ce principe, on peut réduire chaque sous-espace d'états de la $\mathsf{PRU}_{\mathsf{P}-1}$ à $\mathtt{r}_{\mathsf{P}}^{max} + \mathtt{r}_{\mathsf{P}-1}^{max}$ états. Finalement, pour une $\mathsf{PRU}_{\mathsf{p}'}$ de la mission,

[38] Quantité de ressources qui suffit à cette PRU pour obtenir à coup sûr un résultat maximal.

```
1 evalueToutesPRUS :
2 pour p ∈ [P..0] faire
3 |   pour n ∈ [N_p..0] faire
4 |   |   pour Q ∈ Q_{p,n} faire
5 |   |   |   evalueNiveau(S_{Q,p,n})
6 |   |   finpour
7 |   finpour
8 finpour
```

Algorithme 8 : Évaluation des états possibles de la mission, PRU par PRU

la taille de chaque sous-espace $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$ à $\sum_{\mathtt{p}=\mathtt{p}'}^{\mathbf{P}} \mathtt{r}_{\mathtt{p}}^{\mathtt{max}}$.

Note 3 : les deux précédentes notes combinées nous permettent de limiter chaque sous-espace d'états à :

$$|\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}| \quad = \quad \min(\sum_{\mathtt{p}=\mathtt{p}'}^{\mathbf{P}} \mathtt{r}_{\mathtt{p}}^{\mathtt{max}}, \mathtt{r}_{\mathtt{D}}) \tag{5.22}$$

Les deux méthodes de calcul produisent des politiques identiques. Seulement, la deuxième méthode peut créer des états qui ne sont pas forcément accessibles depuis l'état initial avec $\mathtt{r}_{\mathtt{D}}$ ressources. Mais nous avons constaté que dans la plupart des cas, l'algorithme qui explore et construit les états pas à pas finit par construire un espace d'états dont la taille diffère très peu de celui qui est généré par la deuxième méthode.

5.2.3 Inconvénients du contrôle global de la mission

La politique obtenue est statique. Une fois calculée, on ne peut plus rajouter de site à explorer sous peine de devoir recalculer entièrement la stratégie. Ceci n'est pas envisageable pour un robot qui évolue dans un environnement ou des changements peuvent intervenir, et c'est ce que nous voulons intégrer au mécanisme de contrôle de la mission.

Ce qui est important, c'est que le robot ait une politique définie sur la PRU courante, pour pouvoir choisir la bonne action au bon moment. La politique à appliquer en fin de mission n'est pas utile, il suffit de connaître la valeur des états du début de la PRU suivante pour déterminer la politique locale (voir équation 5.10). Nous allons donc dissocier dans la section suivante la fonction de valeur locale (et la politique locale associée) de la fonction de valeur du reste de la mission, et introduire le coût occasionné, qui permet de faire cette séparation.

5.3 Séparer le raisonnement en deux temps

Changeons maintenant légèrement de point de vue. Au lieu de considérer la mission comme un tout, considérons qu'au moment où nous calculons la valeur d'un état **s** nous avons deux parties de l'espace d'états bien distinctes : les états de la PRU courante, et les états des PRUs suivantes. On ne tient pas compte à cet instant des états des PRUs qui sont en amont dans la mission vu que la valeur de l'état courant ne dépend que de la valeur des états suivants.

Nous avons constaté au début de cette partie que : *"il n'est pas nécessaire d'obtenir une solution optimale, mais satisfaisante."* et que *"les ressources nécessaires à la construction de la solution optimale à un problème réduisent l'utilité globale du système."*.

La question peut tout simplement se transformer en celle-ci : *"Puisqu'il me reste* $\mathtt{r}$ *ressources, que vais-je gagner en améliorant la PRU courante, et que vais-je perdre par la suite en n'économisant pas les ressources que je vais utiliser tout de suite ?"*. Ou encore *"Est-ce vraiment utile de dépenser des ressources maintenant ?"*.

Ceci est une interprétation sémantique légèrement différente de l'équation de Bellman 5.9, où la question du max revient à *"Vaut-il mieux exécuter et dépenser des ressources maintenant, ou économiser pour en faire quelque chose de mieux plus tard ?"*, *"***E** *ou* **M** *?"*.

Nous voyons dès à présent qu'en séparant les deux parties *présent - futur*, on va pouvoir mettre en place un système dont le point de vue sera légèrement différent sémantiquement mais dont la solution, sous forme de politique locale à la PRU courante, sera équivalente.

Nous allons donc ne plus récompenser l'agent à la fin de la PRU par le biais de la fonction $\mathcal{R}$, mais remplacer cette récompense par une fonction d'utilité dont nous introduisons enfin le concept.

5.3.1 La fonction d'utilité

L'agent rationnel essaye de maximiser son gain à long terme. Aucun gain n'est obtenu tant qu'une PRU n'est pas terminée. La seule récompense que peut obtenir le robot est donnée via la fonction d'utilité. Elle régit donc le mécanisme de contrôle. Cette fonction permet au final de calculer la valeur d'un état et de la même façon le gain espéré par l'exécution de chaque action, et donc de trouver l'action qu'il faudra associer à cet état.

Chaque $\mathsf{PRU}_\mathtt{p}$ dispose de sa propre fonction d'utilité $\mathbf{U}_\mathtt{p}(\mathbf{Q}, \mathtt{r})$. Cette fonction est une donnée du problème. Elle devra respecter deux hyphothèses :

Hypothèse 1 *La fonction d'utilité est croissante avec les ressources restantes :*

$$\forall \mathbf{Q} \in \mathbb{R}, \forall \mathtt{r}, \mathtt{r}' \in \mathtt{R}, \mathtt{r} < \mathtt{r}' \Rightarrow \mathbf{U}_\mathtt{p}(\mathbf{Q}, \mathtt{r}) \leq \mathbf{U}_\mathtt{p}(\mathbf{Q}, \mathtt{r}')$$

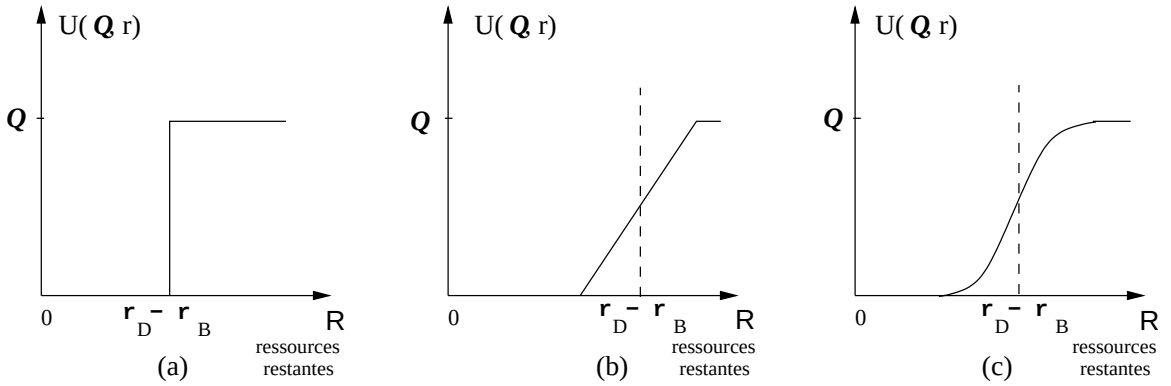


FIG. 5.7 – Fonctions d'utilité

Explications *pour la figure 5.7 :* $\mathbf{U_p}(\mathbf{Q}, \mathtt{r})$ *représente l'utilité d'avoir accumulé une qualité* $\mathbf{Q}$ *dans une PRU sachant qu'il reste* $\mathtt{r}$ *ressources à l'agent. Toutes ces courbes représentent des utilités sachant que la qualité* $\mathbf{Q}$ *est obtenue. La figure (a) représente une date butoir stricte, si l'agent a consommé plus de* $\mathtt{r_B}$ *ressources, c'est-à-dire qu'il lui reste moins de* $\mathtt{r_D} - \mathtt{r_B}$ *ressources, la PRU ne lui rapporte plus rien, sinon, elle lui rapporte* $\mathbf{Q}$*. Les figures (b) et (c) représentent des fonctions d'utilité pour lesquelles le fait de se rapprocher de la date butoir est pénalisé, mais moins fortement qu'en (a). Un agent a la possibilité de finir une tâche avec un léger retard et de trouver une petite utilité dans ces deux cas.*

Hypothèse 2 *La fonction d'utilité est croissante avec la qualité accumulée :*

$$\forall \mathtt{r} \in \mathtt{R}, \forall \mathbf{Q}, \mathbf{Q}' \in \mathbb{R}, \mathbf{Q} < \mathbf{Q}' \Rightarrow \mathbf{U_p}(\mathbf{Q}, \mathtt{r}) \leq \mathbf{U_p}(\mathbf{Q}', \mathtt{r})$$

La fonction d'utilité permet, pour certaines applications, de tenir compte par exemple d'une date butoir $\mathtt{r_B}$ après laquelle le résultat d'une tâche n'aura plus de valeur (sachant que l'agent dispose au départ de $\mathtt{r_D}$ unités de ressource), on peut créer une fonction d'utilité qui vaut $\mathbf{Q}$ tant que la quantité de ressources est supérieure à $\mathtt{r_D} - \mathtt{r_B}$ et 0 sinon (voir figure 5.7). On peut aussi imaginer des fonctions qui décroissent moins brutalement qu'une fonction escalier, comme sur les figures 5.7.b ou 5.7.c.

La fonction d'utilité, qui rappelons-le, va remplacer la fonction de récompense en fin de PRU, est donc le critère à maximiser localement par l'agent rationnel. Afin de satisfaire le même critère que précédemment sur le long terme, c'est-à-dire la somme des qualités accumulées dans toutes les PRUs en tenant compte des ressources consommées, nous allons recréer une fonction d'utilité spécifique.

Nous allons définir l'utilité comme la différence entre ce que l'on gagne en dépensant localement des ressources moins ce que l'on perd en n'économisant pas ce que l'on a consommé. Le coût occasionné, utilisé astucieusement par A.I. Mouaddib et S. Zilberstein est un concept

provenant du monde de la théorie économique.

5.3.2 La technique du coût occasionné.

En économie, le coût occasionné désigne le coût d'une chose estimé en termes d'opportunités non réalisées (et les avantages qui auraient pu êtres retirés de ces opportunités), ou encore la valeur de la meilleure option non réalisée. Plus trivialement, c'est la mesure des avantages auxquels on renonce en affectant les ressources disponibles à un usage donné.

On note $V^*_{\mathtt{p}}$ la fonction de gain espéré $\mathtt{R} \rightarrow \mathbb{R}$ qui vaut la fonction de valeur définie sur le sous-espace d'états $\mathcal{S}_{0,\mathtt{p},\mathbf{0}}$.

Définition 9 *Gain espéré **optimal** : On note V^*_p la fonction qui mesure le gain espéré pour une mission composée des PRUs $[\mathtt{p}, \ldots, \mathbf{P}]$:*

$$\forall \mathbf{r} \in \mathtt{R}, V^*_{\mathtt{p}}(r) \quad = \quad V^*([\mathtt{r}, 0, \mathtt{p}, 0])$$

Pour calculer la politique optimale pour la PRU $\mathtt{p}$, il suffit de connaître $V^*_{\mathtt{p+1}}(r)$. Nous allons remplacer la partie droite des équations 5.5 à 5.8 par une fonction d'utilité qui englobe la fonction de gain espéré. Le contrôle de la PRU $\mathtt{p}$ sera alors lié à la fonction d'utilité suivante, qui donne la valeur des états terminaux de cette PRU :

Définition 10 *Utilité locale :*

$$\mathbf{U}_{\mathtt{p}}(\mathbf{Q}, \mathtt{r}) \quad = \quad \mathcal{R}([\mathtt{r}, \mathbf{Q}, \mathtt{p}, \mathtt{N_p}]) + V^*_{\mathtt{p+1}}(\mathtt{r}) \tag{5.23}$$

Ou plus simplement, puisque ce sont les états du dernier niveau :

$$\mathbf{U}_{\mathtt{p}}(\mathbf{Q}, \mathtt{r}) \quad = \quad \mathbf{Q} + V^*_{\mathtt{p+1}}(\mathtt{r})$$

La récompense obtenue localement tant qu'une PRU n'est pas terminée, c'est-à-dire avant le dernier niveau, reste nulle. Puisque l'on a une stricte égalité entre la définition 10 et les équations 5.5 à 5.8, la politique obtenue en utilisant la fonction utilité sera la même, donc optimale. Nous avons séparé formellement le calcul de la politique locale de la la fonction de valeur pour le reste de la mission.

Notre mécanisme de contrôle va maintenant être séparé en deux parties.

1. Calculer la **fonction de valeur** (optimale) sur **tout** l'espace d'états lié à toutes les PRUs de la mission.
2. Calculer **localement la politique** à chaque nouvelle exécution de PRU grâce à la fonction de gain espéré déjà connue.

La deuxième étape est très rapide, puisqu'elle ne concerne qu'une seule PRU. La première étape prend plus de temps. Mais nous allons montrer dans le chapitre 8 qu'il est possible d'obtenir assez rapidement une estimation grossière de cette fonction de valeur pour le reste de la mission. Nous montrerons donc que les deux étapes peuvent être effectuées instantanément.

Nous notons PRU_0 la PRU que l'agent va exécuter et pour laquelle nous voulons calculer une fonction de valeur locale (et la politique associée). Avant d'exécuter la PRU_0, l'agent dispose d'un certain montant de ressource $\mathtt{r_D}$.

Le coût occasionné OC représente par définition ce que l'agent aurait pu espérer gagner dans le futur, c'est-à-dire $V_1^*(\mathtt{r_D})$, par rapport à ce qu'il pourrait espérer gagner s'il dépense $\Delta\mathbf{r}$ ressources dans la PRU_0, c'est-à-dire $V_1^*(\mathtt{r_D} - \Delta\mathbf{r})$. La valeur espérée pour le reste de la mission se note V_1^* puisque l'agent va commencer la PRU_0.

Définition 11 *Coût occasionné :*

$$OC(\mathtt{r_D}, \Delta\mathbf{r}) = V_1^*(\mathtt{r_D}) - V_1^*(\mathtt{r_D} - \Delta\mathbf{r}) \tag{5.24}$$

On définit alors une nouvelle fonction d'utilité $\mathbf{U}'_0(\mathbf{Q}, \mathbf{r}|\mathtt{r_D})$ qui tient compte du coût occasionné, et des ressources de départ $\mathtt{r_D}$.

Théorème 2 *Soit la fonction d'utilité* $\mathbf{U}'_0(\mathbf{Q}, \mathbf{r})$ *:*

$$\mathbf{U}'_0(\mathbf{Q}, \mathbf{r}|\mathtt{r_D}) = \mathcal{R}([\mathbf{r}, \mathbf{Q}, \mathtt{p}, \mathtt{N_p}]) - OC(\mathtt{r_D}, \mathtt{r_D} - \mathbf{r}) \tag{5.25}$$

Cette fonction d'utilité locale à la PRU courante pénalise maintenant le fait de consommer des ressources contrairement à la première définition de la fonction d'utilité qui était simplement égale à la récompense brute : la somme des qualités accumulées. De ce fait, le contrôle devient maintenant local à la PRU_0, et tient compte non seulement de la qualité accumulée mais aussi des ressources consommées. L'idée de compromis qualité/ressources que nous avons abordée dans le chapitre précédent est bien mise en valeur par le principe du coût occasionné.

Finalement, la fonction d'utilité $\mathbf{U}'_0(\mathbf{Q}, \mathbf{r}|\mathtt{r_D})$ permet de calculer une politique optimale pour la première PRU.

Preuve : La preuve de ce théorème se trouve dans [Mouaddib et Zilberstein, 2000]. La fonction d'utilité de la définition 10 est équivalente à la fonction d'utilité du théorème 2, à une constante près. Puisque le fait d'ajouter une constante à une fonction de récompense n'affecte pas la politique, la politique obtenue par le coût occasionné sera elle aussi optimale. □

En utilisant la programmation dynamique comme nous l'avons fait pour l'algorithme 5, nous évaluons chaque état de la PRU courante, et nous lui affectons une action. Si nous supposons qu'au moment où l'agent commence la PRU_0, la fonction de gain espérée V_1^* est connue, le calcul du coût occasionné ne demande à chaque fois qu'une simple soustraction. Le calcul de la politique locale est toujours aussi rapide.

5.4 Bilan et perspectives

Le raisonnement progressif est en premier lieu un système de description de tâche permettant une double flexibilité :

– la hiérarchie par **niveaux** permet une **interruptibilité** dans l'exécution des tâches

– L'approche **modulaire** permet d'**adapter la consommation** de ressources.

Le formalisme proposé permet de modéliser assez simplement les tâches à exécuter, avec une certaine souplesse. Cependant, la politique permettant de maximiser le critère de gain de l'agent rationnel est longue à calculer. Ceci est dû a une représentation de l'état (PRU, niveau, qualité, ressource) qui engendre un espace d'états de grande taille quand le nombre de PRUs vient à augmenter.

La séparation du raisonnement présent et de l'espérance future nous permet d'envisager de ne pas forcément calculer la fonction de valeur sur tous les états futurs mais seulement sur ceux qui sont accessibles. En fait, il suffit d'avoir la valeur des états du niveau 0 de la PRU suivante pour calculer la fonction de valeur de la PRU courante. On a remplacé la fonction de récompense immédiate obtenue après la fin d'une PRU par une fonction d'utilité, compromis entre la qualité obtenue localement et le coût occasionné par les ressources consommées.

La séparation de l'approche va permettre plusieurs choses. Nous allons utiliser le concept bien connu de *"diviser pour régner"* pour trouver une solution élégante à notre problème, divisé en deux parties :

1. calculer la fonction de valeur pour le reste de la mission (le **futur**),
2. calculer la fonction de valeur locale (donc la politique à exécuter localement) connaissant cette fonction de valeur (le **présent**).

Le problème ainsi divisé nous laisse la possibilité d'apporter une amélioration majeure à ce qui à déjà été fait : la gestion d'un imprévu dans la mission, obligeant à reconsidérer complètement le plan existant devient envisageable : la politique locale est calculée très rapidement si on connaît la fonction de valeur pour le futur, et si nous réussissons à estimer rapidement cette fonction de valeur (par le biais d'approximations), nous obtiendrons une fonction de valeur locale non optimale qui conduira à une politique locale non optimale, générant ainsi un comportement permettant de s'adapter provisoirement au changement imprévu. Mais le comportement non optimal ne sera pas si différent de celui d'un agent purement rationnel. Nous nous y attelons dans les chapitres 7 et 8.

Conclusion

Nous avons présenté dans ce chapitre le formalisme du raisonnement progressif tel que l'avaient pensé A.I. Mouaddib et S. Zilberstein. Ce formalisme se prête bien à l'exploration

planétaire et à la robotique autonome. Chaque mission peut être décrite graduellement par un expert scientifique qui pourra pondérer les résultats qu'il attend avec le système de qualité, et gérer l'incertitude de consommation de ressources grâce aux probabilités. L'exécution des tâches peuvent se faire pas à pas, grâce aux niveaux, et des alternatives sont possibles grâce aux modules. Nous présentons d'ailleurs une application concrète de ce modèle dans le chapitre 9.

L'algorithme de programmation dynamique par exploration de l'espace d'états permet d'obtenir une politique optimale pour la mission affectée au robot. Cette politique est cependant longue à calculer, car son temps de calcul est proportionnel à l'espace d'états, qui lui augmente au carré avec le nombre de PRUs.

Le chapitre suivant propose une façon d'intégrer les ressources multiples au raisonnement progressif et d'adapter le calcul de la fonction de valeur de façon à ce que l'on obtienne une fonction de valeur optimale tenant compte de ressources multiples dans un temps raisonnable.

Dans la partie suivante nous tenterons d'estimer rapidement les valeurs que peut prendre la fonction de coût occasionné, de façon à ce que l'on puisse obtenir rapidement une politique locale approchée. Cette politique approchée pourra servir provisoirement en cas d'imprévus dans l'environnement qui obligerait le robot à changer de mission.

Chapitre 6

Passage aux ressources multiples

Introduction

Un robot évoluant dans un environnement réel dispose de plusieurs ressources consommables : l'énergie présente dans ses batteries, la mémoire pour stocker les informations récoltées, le temps imparti pour effectuer la mission.

Même si les robots martiens actuels (Spirit et Opportunity) possèdent des panneaux solaires qui leur permettent de se recharger en permanence, il n'est pas exclu que ceux-ci soient recouverts de poussière. Il se peut également que le niveau des batteries soit faible pendant la nuit. Il faut dans ces deux cas traiter l'énergie comme une ressource consommable.

Le modèle proposé par A.I. Mouaddib et S. Zilberstein ne permet de tenir compte que d'une ressource, le temps. Mais il laisse pourtant la possibilité d'être étendu aux ressources multiples. Dans ce chapitre nous allons continuer à exploiter le formalisme du raisonnement progressif en y intégrant ces ressources multiples. Cependant, la taille de l'espace des états possibles de la mission augmente à cause de la cardinalité supplémentaire imposée par les autres ressources.

Les algorithmes présentés dans le chapitre précédent peuvent fournir une politique de contrôle optimale pour la première PRU. Le temps de calcul nécessaire pour l'obtenir de cette façon est en relation avec la taille de l'espace d'états, et est plus long qu'avec une seule ressource. L'introduction de nouvelles ressources apporte de nouveaux problèmes, plus connus sous le nom de « malédiction de la dimension » (*« curse of dimensionnality »* en anglais). Cette caractéristique est relative à la programmation dynamique [Bellman, 1957]. Le temps et la mémoire nécessaires pour calculer notre fonction de valeur croît exponentiellement avec le nombre de variables nécessaires pour décrire l'état. Nous proposons dans ce chapitre un algorithme adapté aux ressources multiples, basé sur le principe d'agrégation de l'espace d'états.

Notre apport pour le contrôle de la consommation de ressources multiples dans le cadre du raisonnement progressif est double :

1. un formalisme étendu pour les PRUs et le mécanisme de contrôle afin de pouvoir traiter de multiples ressources,
2. un algorithme plus rapide pour calculer la fonction de valeur exacte sur tous les états possibles de la mission.

Nous avons également mis au point une représentation compacte du domaine de la fonction de gain espéré pour gagner de l'espace mémoire. Certains résultats de ce chapitre ont donné lieu à des publications internationales [Le Gloannec *et al.*, 2004, Le Gloannec *et al.*, 2005b].

6.1 Formalisme étendu à plusieurs ressources

Notre premier apport est l'extension du formalisme du raisonnement progressif à plusieurs ressources. L'extension se base sur ce qui a été présenté dans le chapitre précédent, et des modifications sont effectuées pour :

- la description des ressources,
- les unités de raisonnement progressif,
- le MDP pour le contrôle, notamment les états.

Nous présentons ces extensions dans cette section, avant de présenter dans la section suivante des algorithmes adaptés aux ressources multiples par programmation dynamique dans le cadre du contrôle de mission avec le raisonnement progressif.

6.1.1 Les ressources multiples

Nous avons l'ambition d'étendre ce modèle à plusieurs ressources consommables, pour que le robot explorateur autonome puisse tenir compte de toutes les ressources embarquées (temps, énergie, espace mémoire etc ...).

Notre extension du modèle est la suivante : notre ressource unique $\mathtt{r}$ devient le n-uplet de ressources $\overline{\mathtt{r}} = (\mathtt{r}_1, \ldots, \mathtt{r}_\omega, \ldots, \mathtt{r}_\Omega)$ où chaque ressource $\mathtt{r}_\omega$ est considérée comme discrète, et son montant est compris entre 0 et le montant de départ noté $\mathtt{r}^{\mathtt{D}}_\omega$. Certaines de ces ressources sont en réalité continues. Nous choisissons de les discrétiser.

L'espace des ressources consommables $\mathtt{R}$ devient quant à lui un produit cartésien $\bigotimes_{\omega=1}^{\Omega} \left(\mathtt{R}_\omega \right)$.

Nous définissons un opérateur d'ordre partiel de dominance sur $\mathtt{R}$ « $\prec$ ». Cet opérateur nous indique si un n-uplet de ressources contient plus de ressources qu'un autre n-uplet de ressources.

Définition 12 *Ordre partiel de dominance*

Soient deux n-uplets de ressources $\overline{\mathtt{r}^{\mathtt{a}}}$ *et* $\overline{\mathtt{r}^{\mathtt{b}}}$*, nous définissons* $\prec$ *la relation de dominance entre ces deux n-uplets comme :*

$$\forall \omega, \mathtt{r}^{\mathtt{a}}_\omega \leq \mathtt{r}^{\mathtt{b}}_\omega \text{ et } \exists \omega', \mathtt{r}^{\mathtt{a}}_{\omega'} < \mathtt{r}^{\mathtt{b}}_{\omega'} \quad \text{alors} \quad \overline{\mathtt{r}^{\mathtt{a}}} \prec \overline{\mathtt{r}^{\mathtt{b}}}.$$

$$\forall \omega, \mathtt{r}^{\mathtt{a}}_{\omega} \leq \mathtt{r}^{\mathtt{b}}_{\omega} \quad alors \quad \overline{\mathtt{r}^{\mathtt{a}}} \preceq \overline{\mathtt{r}^{\mathtt{b}}}.$$

6.1.2 Ressources bornées et décroissantes

Les différentes ressources restantes qui définissent le n-uplet de ressources sont décroissantes et bornées : nous supposons qu'au début de la mission, le robot dispose d'un certain montant de ressources $\overline{\mathtt{r}^{\mathtt{D}}}$, et module après module, niveau après niveau, PRU après PRU, la quantité de chaque ressource $\mathtt{r}_{\omega}$ diminue sous l'effet des actions sélectionnées dans les modules. Chaque ressource $\mathtt{r}_{\omega}$ est donc bornée sur un intervalle de $\mathtt{R}_{\omega}$ entre $\mathtt{r}^{\mathtt{D}}_{\omega}$ et 0. Nous choisissons dès à présent de simplifier les notations en confondant $\mathtt{R}_{\omega}$ avec l'intervalle $[0 \ldots \mathtt{r}^{\mathtt{D}}_{\omega}]$.

$$\forall \omega \in [1, \ldots, \Omega], \mathtt{R}_{\omega} \quad = \quad [0, \ldots, \mathtt{r}^{\mathtt{D}}_{\omega}] \tag{6.1}$$

6.1.3 Granularité de chaque ressources

Le nombre de valeurs que pourra prendre chaque $\mathtt{r}_{\omega}$ sur $\mathtt{R}_{\omega}$ dépend de la granularité que nous donnons à cette ressource. Plus le grain sera fin, plus il y aura de valeurs possibles pour cette ressource, et plus le domaine $\mathtt{R}_{\omega}$ tendra vers le continu. Nous notons Γ_{ω} la granularité de $\mathtt{R}_{\omega}$ qui représentera le nombre de valeurs que pourra prendre la ressource $\mathtt{r}_{\omega}$ sur $\mathtt{R}_{\omega}$.

Pour la ressource temporelle par exemple : considérons que nous disposions d'une heure pour effectuer la mission, donc 3600 secondes ou 60 minutes, si nous choisissons une granularité de l'ordre de la minute ($\Gamma_{\mathtt{t}} = 60$), nous aurons une valeur possible par minute, et le contrôle du temps restant pour la mission se fera à une précision de l'ordre de la minute. Si nous choisissons une granularité de l'ordre de la seconde $\Gamma_{\mathtt{t}} = 3600$, nous pouvons contrôler la mission à la seconde près.

Nous laissons le choix de cette granularité aux contrôleurs du robot, mieux à même de savoir à quel niveau l'incertitude de consommation de chaque ressource se situe. Nous considérerons donc les Γ_{ω} comme des données du problème, au même titre que le n-uplet de ressources initialement disponibles $\overline{\mathtt{r}^{\mathtt{D}}}$.

6.1.4 Dépendance entre les différentes ressources

Nous supposerons que l'agent dispose pour chaque module d'une distribution de probabilités de consommation de ressources jointe. La consommation de chaque ressource sera donc dépendante des autres. Même si la figure 6.1 représente une distribution de probabilités de consommation jointe régulière, nous ne faisons aucune hypothèse particulière sur cette régularité.

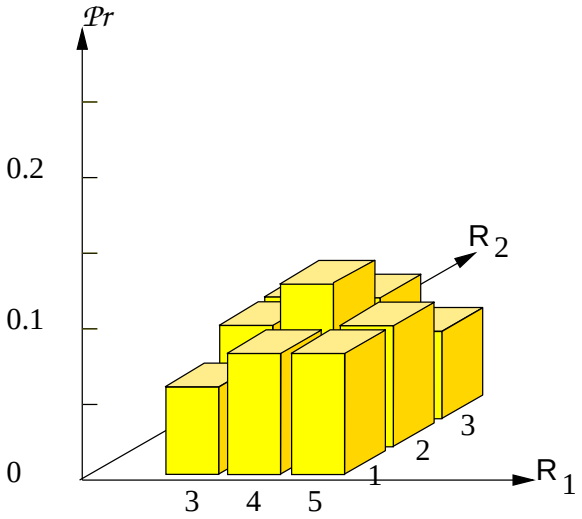


FIG. 6.1 – Distribution de probabilités de consommation jointe de ressources multiples discrétisée.

La distribution de probabilités de consommation de ressources dépend maintenant de plusieurs variables. Notre exemple est illustré avec deux ressources, le temps ($\mathtt{R}_1$), et l'énergie ($\mathtt{R}_2$) consommées par un module donné (disons le module « ramasser une roche »). Dans cet exemple, en ramassant une roche, le robot à une probabilité inférieure à $0,1$ *de consommer* 5 *unités de temps* ***et*** 1 *unité d'énergie. La consommation d'énergie est jointe à la consommation de temps : ici, c'est la somme de toutes les probabilités de consommation qui est égale à* 1.

6.1.5 Description des PRUs

La description d'une PRU reste la même. Chaque PRU est donc partitionnée en niveaux et chaque niveau est subdivisé en modules, qui sont des alternatives pour exécuter différemment chaque niveau. Il y a toujours une qualité mono critère associée à chaque module.

Cependant, la fonction de distribution de probabilités de consommation de ressources est modifiée. Puisqu'il y a passage aux ressources multiples, le domaine de chaque fonction $\mathcal{P}r_{\mathtt{p,n,m}}$: $\mathtt{R} \rightarrow [0,1]$ rattachée à un module $\mathsf{m}_{\mathtt{p,n,m}}$ est inclus dans un espace multidimensionnel $\mathtt{R} = \bigotimes_{\omega=1}^{\Omega} \left(\mathtt{R}_\omega\right)$. La figure 6.1 donne un exemple de fonction $\mathcal{P}r$.

6.2 Mécanisme de contrôle avec plusieurs ressources

Nous avons présenté dans le chapitre précédent comment contrôler la consommation de ressource (une ressource unique, le temps) dans la PRU courante en fonction du coût occasionné par cette consommation sur le reste de la mission. Lorsque plusieurs ressources doivent être contrôlées, le mécanisme de contrôle reste le même, nous séparons :

– le calcul de la fonction de valeur (et de la politique) pour la première PRU,

– le calcul de la fonction de gain espéré pour le reste de la mission.

Le calcul du gain espéré va permettre de calculer la fonction de coût occasionné. Les deux calculs se basent toujours sur la modélisation de la mission par un processus décisionnel de Markov, avec un espace d'états modifié par l'introduction de nouvelles ressources.

6.2.1 Le MDP pour le contrôle de ressources multiples

Le MDP qui nous permet de calculer la politique à exécuter par l'agent lors de sa mission est modifié pour s'adapter aux ressources multiples. Nous conservons la forme de quadruplet $\{\mathcal{S}, \mathcal{A}, \mathcal{P}r, \mathcal{R}\}$.

Les états

La ressource simple $\mathtt{r}$ est transformée en n-uplet de ressources $\overline{\mathtt{r}}$. Nous augmentons donc la cardinalité de chaque état, donc la taille de l'espace d'états augmente.

L'agent atteint un état d'échec lorsqu'un module consomme plus de ressources que le montant disponible avant l'exécution. L'épuisement d'une seule ressource suffit à déclencher un échec. Donc :

$$\exists \omega \in [1, \ldots, \Omega], \mathtt{r}_\omega < 0 \Leftrightarrow \qquad \mathbf{s} = \mathbf{s}_{\mathtt{echec}} \tag{6.2}$$

Formellement, l'espace d'états est : $\mathcal{S} = \{[\overline{\mathtt{r}}, \mathbf{Q}, \mathtt{p}, \mathtt{n}], \overline{\mathtt{r}} \in \mathtt{R}\} \cup \{\mathbf{s}_{\mathtt{echec}}\}$

Les actions de l'agent, et la fonction de transition et de récompense

Les actions sont les mêmes, **E** et **M**, qu'il y ait une ou plusieurs ressources. Les fonctions de transitions sont exprimées en fonction des distributions de probabilité de consommation de ressources. Le principe général ne change pas, l'action **M** reste déterministe, l'action **E** reste probabiliste. L'agent reçoit sa récompense seulement après avoir effectué toute une PRU. Cette récompense est égale à la qualité accumulée dans cette PRU, suivant les modules exécutés.

6.2.2 Etude de complexité

La complexité de la résolution du problème de contrôle est proportionnelle à la taille de l'espace d'états. C'est pourquoi nous allons procéder à un dénombrement de cet espace d'états, en fonction du nombre de ressources allouées et du nombre de PRU dans la mission. Le nombre d'états augmente avec la granularité du domaine de chaque ressource. La taille totale de l'espace des ressources est :

$$|\mathtt{R}| = \prod_{\omega=1}^{\Omega} \left(\Gamma_\omega \right) \tag{6.3}$$

Donc en combinant ceci avec la taille de l'espace d'états exprimée dans l'équation 5.16, nous obtenons un espace d'états $\mathcal{S}$ total de taille bornée par :

$$|\mathcal{S}| \leq \underbrace{\prod_{\omega=1}^{\Omega}\left(\Gamma_{\omega}\right)}_{\text{espace des ressources}} \times \underbrace{\left(\sum_{\mathtt{p}=1}^{\mathbf{P}}\left(1+\sum_{\mathtt{n}=1}^{\mathtt{N_p}}\prod_{\mathtt{n}'=1}^{\mathtt{n}}|\mathtt{M_{p,n'}}|\right)\right)}_{\text{nombre de sous espaces d'états } \mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}} \tag{6.4}$$

La première partie de l'équation représente toutes les ressources qui peuvent rester à l'agent après un niveau, la deuxième partie représente les différentes qualités accumulées possibles après chaque niveau (+1 pour le niveau 0), c'est à dire l'ensemble des sous espaces d'états $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$, ceci sommé sur toutes les PRUs de la mission.

La taille de l'espace d'états va donc dépendre :

- du nombre de ressources Ω,
- de la granularité de chaque ressource Γ_{ω},
- du nombre de PRUs dans la mission $\mathbf{P}$,
- du nombre de niveaux et de modules dans chaque PRU.

La granularité de chaque ressource va jouer un rôle primordial pour la taille de l'espace d'états. En choisissant de représenter le temps sur une échelle de 2 secondes par exemple, l'espace d'états sera deux fois moins grand qu'en choisissant de le représenter sur une échelle d'une seule seconde. En divisant la granularité de chaque ressource par 2, l'espace d'états devient 2^{Ω} fois moins grand. En considérant que toutes les PRUs de la mission sont les mêmes, qu'il y a $\mathbf{C}$ sous espaces d'états $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$ dans ces PRUs et qu'elles consomment au plus $\overline{\mathtt{r}}_{\mathbf{P}}^{max}$ pour chaque PRU, nous obtenons une première borne de la taille de l'espace d'états.

$$|\mathcal{S}| \leq \mathbf{P}.\Big(\underbrace{\mathbf{P}.\mathtt{r}_{\mathbf{P}}^{\mathtt{max}}}_{\mathtt{r}_{mission}^{max}}\Big)^{\Omega} = \mathbf{P}^{\Omega+1}.\left(\mathtt{r}_{\mathbf{P}}^{\mathtt{max}}\right)^{\Omega}.\mathbf{C} \tag{6.5}$$

La taille de l'espace d'états augmente exponentiellement avec le nombre de ressources différentes, comme le montre l'équation 6.5, conformément au principe de malédiction de la dimension de Bellman.

Nous pouvons également réduire cet espace d'états comme nous l'avons fait avec l'équation 5.18 du chapitre précédent, en tenant compte du fait que dans la dernière PRU, l'agent ne pourra pas consommer plus que $\overline{\mathtt{r}}_{\mathbf{P}}^{max}$, $\overline{\mathtt{r}}_{\mathbf{P}}^{max} = (\mathtt{r}_{1\mathbf{P}}^{max},\ldots,\mathtt{r}_{\omega\mathbf{P}}^{max},\ldots,\mathtt{r}_{\Omega\mathbf{P}}^{max})$. Pour chaque ressource prise séparément $\mathtt{r}_{\omega}$ l'agent ne peut pas dépenser plus que $\mathtt{r}_{\omega\mathbf{P}}^{max}$. De ce fait :

$$\begin{aligned} \forall \overline{\mathtt{r}} = (\mathtt{r}_1, \ldots, \mathtt{r}_\omega, \ldots, \mathtt{r}_\Omega), \qquad & \forall \omega, \mathtt{r}_\omega > \mathtt{r}^{max}_{\omega \mathbf{P}}, \\ V([(\mathtt{r}_1, \ldots, \mathtt{r}_\omega, \ldots, \mathtt{r}_\Omega), \mathbf{Q}, \mathtt{p}, \mathtt{n}]) \quad = \quad & V([(\mathtt{r}_1, \ldots, \mathtt{r}^{max}_{\omega \mathbf{P}}, \ldots, \mathtt{r}_\Omega), \mathbf{Q}, \mathtt{p}, \mathtt{n}]) \end{aligned}$$

Nous allons considérer dans la dernière $\mathsf{PRU}_{\mathbf{P}}$ que tous les états dont une des ressources dépasse le montant nécessaire pour tout effectuer seront regroupés avec l'état où cette ressource est présente en quantité maximale et suffisante comme l'exprime l'équation suivante.

$$\forall \omega, [(\mathtt{r}_1, \ldots, \mathtt{r}_\omega > \mathtt{r}^{max}_{\omega \mathbf{P}}, \ldots, \mathtt{r}_\Omega), \mathbf{Q}, \mathtt{p}, \mathtt{n}] \quad = \quad [(\mathtt{r}_1, \ldots, \mathtt{r}^{max}_{\omega \mathbf{P}}, \ldots, \mathtt{r}_\Omega), \mathbf{Q}, \mathtt{p}, \mathtt{n}]$$

De ce fait, les sous espaces d'états de la dernière $\mathsf{PRU}_{\mathbf{P}}$ sont bornés par le n-uplet $\overline{\mathtt{r}}^{max}_{\mathbf{P}}$. De la même façon que dans le chapitre précédent, quand l'agent arrive à la PRU d'indice $\mathtt{p} < \mathbf{P}$, le nombre total de ressources qu'il pourra dépenser par la suite sera inférieur à $\overline{\mathtt{r}^{\mathtt{max}}_{\mathtt{p}}} + \overline{\mathtt{r}^{\mathtt{max}}_{\mathtt{p+1}}} + \cdots + \overline{\mathtt{r}^{\mathtt{max}}_{\mathbf{P}}}$. La taille de l'espace d'états d'une $\mathsf{PRU}_{\mathtt{p}}$ quelconque est donc borné par :

$$|\mathcal{S}_{\mathtt{p}}| \quad \leq \quad \left((\prod_{\omega=1}^{\Omega} \sum_{\mathtt{p}'=\mathtt{p}}^{\mathbf{P}} \mathtt{r}^{\mathtt{max}}_{\omega \mathtt{p}'}) \times (1 + \sum_{\mathtt{n}=1}^{\mathtt{N_p}} \prod_{\mathtt{n}'=1}^{\mathtt{n}} |\mathtt{M}_{\mathtt{p},\mathtt{n}'}|) \right)$$

Et pour toutes les PRUs de la mission, la taille de l'espace d'état est borné par :

$$|\mathcal{S}| \quad \leq \quad \sum_{\mathtt{p}=1}^{\mathbf{P}} \left((\prod_{\omega=1}^{\Omega} \sum_{\mathtt{p}'=\mathtt{p}}^{\mathbf{P}} \mathtt{r}^{\mathtt{max}}_{\omega \mathtt{p}'}) \times (1 + \sum_{\mathtt{n}=1}^{\mathtt{N_p}} \prod_{\mathtt{n}'=1}^{\mathtt{n}} |\mathtt{M}_{\mathtt{p},\mathtt{n}'}|) \right) \tag{6.6}$$

Nous allons considérer comme précédemment que toutes les PRUs sont de même type et que les ressources ont toutes le même domaine de définition :

$$|\mathcal{S}| \quad \leq \quad (\sum_{\mathtt{p}=1}^{\mathbf{P}} \mathtt{p}^{\Omega}) . (\mathtt{r}^{\mathtt{max}}_{\mathbf{P}})^{\Omega} . \mathbf{C},$$

où $\mathbf{C}$ est le nombre de sous espaces d'états $\mathcal{S}_{\mathbf{Q},\mathbf{P},\mathtt{n}}$ dans chaque PRU. Cette borne est plus fine que celle donnée par 6.5. Avec une seule ressource, nous divisions (en moyenne) l'espace d'états par deux, ici, la réduction est encore plus forte. Pour 2 ressources différentes, la réduction est de 66% ; pour 3 ressources on obtient 75% d'états en moins. Cette réduction est d'autant plus intéressante qu'il y a de ressources.

6.2.3 Le contrôle de la première PRU

Comme le contrôle de la mission se fait en deux temps, nous expliquons ici comment contrôler l'exécution de la première PRU en supposant que le calcul de la fonction de gain espéré a déjà été calculée. Le calcul de cette fonction sera l'objet de la section 6.3. La fonction d'utilité, qui en dépend, est la base du calcul de la politique de contrôle.

La fonction d'utilité

La fonction d'utilité, qui dépend des ressources restantes, est croissante :

Hypothèse 3 *La fonction d'utilité est croissante selon les ressources multiples restantes :*

$$\forall \mathbf{Q} \in \mathbb{R}, \forall \mathtt{r}, \mathtt{r}' \in \mathtt{R}, \mathtt{r} \prec \mathtt{r}' \Rightarrow \mathbf{U}_{\mathtt{p}}(\mathbf{Q}, \mathtt{r}) \leq \mathbf{U}_{\mathtt{p}}(\mathbf{Q}, \mathtt{r}') \tag{6.7}$$

Grâce à cette fonction d'utilité, définie cette fois-ci avec plusieurs ressources, nous pouvons calculer exactement de la même façon une politique de contrôle optimale[39] pour la PRU courante.

Calcul de la politique de contrôle

Le principe de contrôle restant le même, la définition de la fonction de valeur pour la PRU locale reste aussi la même. Nous supposons dans ce paragraphe que la fonction d'utilité a déjà été calculée, nous n'exposons ici que les équations qui permettent de calculer la politique de contrôle pour la première PRU. Le calcul de la fonction de gain espérée optimale sur l'ensemble des PRUs restantes de la mission sera largement détaillé dans la suite de ce chapitre.

Nous redonnons les équations de cette fonction de valeur :

$$V([\overline{\mathtt{r}}, \mathbf{Q}, \mathtt{p}, \mathtt{N_p}]) \quad = \quad \mathbf{U}(\mathbf{Q}, \overline{\mathtt{r}}) \tag{6.8}$$

La valeur d'un état avant le dernier niveau dépend de la valeur espérée en exécutant les niveaux supérieurs, et en choisissant le meilleur module.

$$V_{\mathbf{E}}([\overline{\mathtt{r}}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]) \quad = \quad \max_{\mathtt{m}} EV(\mathbf{E}_{\mathtt{m}}, [\overline{\mathtt{r}}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]) \tag{6.9}$$

ou EV est une fonction de valeur espérée qui s'écrit de la façon suivante :

$$EV(\mathbf{E}_{\mathtt{m}}, [\overline{\mathtt{r}}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]) =$$

$$\underbrace{\sum_{\mathbf{Q}', \Delta\overline{\mathtt{r}} > \overline{\mathtt{r}}} \mathcal{P}r_{\mathtt{p},\mathtt{n}+1,\mathtt{m}}((\mathbf{Q}', \Delta\overline{\mathtt{r}}) | \, \mathbf{Q}) \times V(\mathbf{s}_{\mathtt{echec}})}_{\text{Si une ressource est épuisée}} + \underbrace{\sum_{\mathbf{Q}', \Delta\overline{\mathtt{r}} \leq \overline{\mathtt{r}}} \mathcal{P}r_{\mathtt{p},\mathtt{n}+1,\mathtt{m}}((\mathbf{Q}', \Delta\overline{\mathtt{r}}) | \, \mathbf{Q}) \times V([\overline{\mathtt{r}} - \Delta\overline{\mathtt{r}}, \mathbf{Q} + \mathbf{Q}', \mathtt{p}, \mathtt{n}+1])}_{\text{Si le module est exécuté}}$$

[39] Le critère est toujours de maximiser le gain espéré.

Pour contrôler la première PRU, il suffit de rajouter l'équation suivante, liée à la valeur pour l'action changement de PRU $\mathbf{M}$:

$$V_{\mathbf{M}}([\overline{\mathtt{r}}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]) \quad = \quad \mathbf{U}(\mathbf{Q}, \overline{\mathtt{r}}) \tag{6.10}$$

Finalement, en gardant la meilleure action, nous retrouvons encore notre équation de Bellman, adaptée au contexte du raisonnement progressif.

$$V([\overline{\mathtt{r}}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]) \quad = \quad \max(V_{\mathbf{M}}, V_{\mathbf{E}}) \tag{6.11}$$

6.2.4 Calcul de la fonction de gain espéré

Nous pouvons toujours utiliser les deux algorithmes (5 et 7) vus dans le chapitre précédent pour calculer la fonction de gain espéré pour toutes les PRUs de la mission (sauf la première) afin d'obtenir la fonction de coût occasionné et donc la fonction d'utilité pour la politique de contrôle optimale avec plusieurs ressources dans la première PRU. Seulement, avec l'augmentation du nombre de ressources apparaît le phénomène de malédiction de la dimension : l'augmentation du nombre de variables d'états fait croître le temps de calcul de la fonction de gain espéré. Nous avons donc adapté les algorithmes de programmation dynamique pour calculer la fonction de gain espéré avec des ressources multiples afin de limiter l'effet de l'explosion de l'espace d'états.

6.3 Adaptation de la programmation dynamique aux ressources multiples

Notre formalisme étendu nous permet d'exprimer les incertitudes relatives à la consommation de ressources multiples. Cependant, la taille de l'espace d'états pour une mission donnée croît exponentiellement avec le nombre de ressources différentes, et au carré avec le nombre de *PRU* dans la mission. Des expériences menées avec les algorithmes initiaux (5 et 7) nous montrent[40] que pour une mission comprenant une dizaine de PRUs, avec chacune deux à trois niveaux et deux ou trois modules consommant deux ressources différentes, il faut plusieurs minutes pour calculer la politique optimale ; nous voudrions obtenir le même résultat plus rapidement.

Notre objectif est le suivant : **limiter la taille de l'espace d'états** pour obtenir plus rapidement la politique de contrôle optimale. Pour cela, nous allons utiliser la séparation du raisonnement en deux parties :

1. le calcul d'une fonction de gain espéré pour toutes les PRUs après la première,

[40] Voir chapitre 10 : validation des résultats.

2. le calcul de la politique de la PRU courante en fonction du gain espéré.

Les améliorations que nous avons apportées pour le contrôle d'une mission via le raisonnement progressif étendu à plusieurs ressources se basent sur une exploitation de la structure de l'espace d'états et sur des propriétés mathématiques de la fonction de valeur.

Les améliorations exposées dans cette section ne suivent pas l'ordre chronologique de nos résultats de thèse. En effet, au cours de celle-ci, nous avons développé plusieurs idées qui nous ont mené à l'algorithme final d'**agrégation faible** que nous vous présenterons en section 6.3.3 page 109. Notre ligne de conduite générale a toujours été dans ce contexte d'agréger l'espace d'états pour contrecarrer l'effet de la malédiction de la dimension. Nous avons donc naturellement commencé par essayer d'agréger le plus possible cet espace d'état, par une méthode d'**agrégation forte** (résultats publiés dans [Le Gloannec *et al.*, 2004, Le Gloannec *et al.*, 2005b]) présentée en fin de chapitre. Mais cette agrégation n'apportait pas suffisamment d'améliorations, et nous avons donc abandonné provisoirement l'idée d'agrégation pour exploiter certaines **propriétés de la fonction de valeur**. Finalement, ces propriétés nous ont permis de trouver la méthode d'agrégation faible.

Dans ce qui suit, nous allons exposer dans un premier temps certaines propriétés de la fonction de valeur qui nous ont été utiles pour améliorer l'algorithme de programmation dynamique basique. Un sens de parcours adapté à l'espace d'états a été adopté pour accélérer l'algorithme de programmation dynamique. Nous verrons ensuite comment réutiliser ce sens de parcours pour créer une structure agrégée de l'espace d'états, que nous avons appelée agrégation faible. Finalement, nous présentons la méthode d'agrégation forte, et ses inconvénients.

6.3.1 Propriétés de la fonction de valeur V

Nous avons classé certaines propriétés de la fonction de valeur V afin de pouvoir les exploiter dans un algorithme de programmation dynamique adapté. La fonction de valeur est croissante, et certaines ressources sont un facteur limitant à cette croissance.

Croissance de la fonction de valeur

Sur un sous espace d'états $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$ donné, la fonction de valeur V est croissante avec les ressources disponibles pour une qualité accumulée donnée et à un niveau donné.

Propriété 1

$$\overline{\mathtt{r}^{\mathtt{a}}} \prec \overline{\mathtt{r}^{\mathtt{b}}} \Rightarrow V([\overline{\mathtt{r}^{\mathtt{a}}}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]) < V([\overline{\mathtt{r}^{\mathtt{b}}}, \mathbf{Q}, \mathtt{p}, \mathtt{n}])$$

Cette propriété découle du fait que $\mathbf{U}$ est croissante.

Remarque : pour des raisons de clarté dans l'explication de ce qui va suivre, nous allons énoncer nos équations avec seulement deux ressources. Elles restent cependant généralisables à plus de 2 ressources consommables.

Ressource limitante et paliers dans la fonction de valeur

Nous avons repéré des paliers (voir figure 6.2) dans la fonction de valeur sur un sous espace d'états $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$ donné. Une partie de ces paliers dans la fonction de valeur est due à une **ressource limitante**, c'est-à-dire une ressource présente en quantité insuffisante.

Avec deux ressources, deux seuils sont nécessaires pour effectuer un module, tant qu'une des deux ressources n'est pas présente en quantité suffisante, la valeur stagne (voir figure 6.2). Si par exemple il faut au moins 4 secondes et 8 unités d'énergie pour ramasser une roche, et que l'agent dispose de 3 secondes, alors la quantité d'énergie disponible n'y changera rien : même si elle est infinie, l'agent est limité par le temps. Le temps agit dans cet exemple comme une ressource critique, qui limite le gain espéré.

Nous formalisons le phénomène de facteur limitant dans les remarquess suivantes que nous exploitons pour accélérer le calcul de la fonction de valeur sur un sous espace d'état donné.

Remarque

$$\begin{aligned}\forall(\mathtt{r}_1,\mathtt{r}_2)\in\mathtt{R},\forall\mathbf{Q},\mathtt{p},\mathtt{n},0\leq V([(\mathtt{r}_1,\mathtt{r}_2),\mathbf{Q},\mathtt{p},\mathtt{n}]) &\leq V([(\mathtt{r}_1,+\infty),\mathbf{Q},\mathtt{p},\mathtt{n}])\\ \forall(\mathtt{r}_1,\mathtt{r}_2)\in\mathtt{R},\forall\mathbf{Q},\mathtt{p},\mathtt{n},0\leq V([(\mathtt{r}_1,\mathtt{r}_2),\mathbf{Q},\mathtt{p},\mathtt{n}]) &\leq V([(+\infty,\mathtt{r}_2),\mathbf{Q},\mathtt{p},\mathtt{n}])\end{aligned}$$

Où $+\infty$ représente une quantité théoriquement infinie de ressources (de type 1 ou 2). Mais, pour une quantité $\mathtt{r}_1$ donnée, cette valeur $V([(\mathtt{r}_1,+\infty),\mathbf{Q},\mathtt{p},\mathtt{n}])$ peut être atteinte pour une quantité $\mathtt{r}_2$ allouée. $\mathtt{r}_1$ est alors le facteur limitant et nous remarquons la chose suivante :

Remarque

$$\begin{aligned}&\forall\mathbf{Q},\mathtt{p},\mathtt{n},\forall\mathtt{r}_1\in\mathtt{R}_1,\exists\mathtt{r}_2\in\mathtt{R}_2 \quad \text{tel que} \quad V([(\mathtt{r}_1,\mathtt{r}_2),\mathbf{Q},\mathtt{p},\mathtt{n}])=V([(\mathtt{r}_1,+\infty),\mathbf{Q},\mathtt{p},\mathtt{n}]),\\ \Rightarrow\ &\forall\mathtt{r}_2'\geq\mathtt{r}_2, V([(\mathtt{r}_1,\mathtt{r}_2'),\mathbf{Q},\mathtt{p},\mathtt{n}]) = V([(\mathtt{r}_1,+\infty),\mathbf{Q},\mathtt{p},\mathtt{n}])\\ &\forall\mathbf{Q},\mathtt{p},\mathtt{n},\forall\mathtt{r}_2\in\mathtt{R}_2,\exists\mathtt{r}_1\in\mathtt{R}_1 \quad \text{tel que} \quad V([(\mathtt{r}_1,\mathtt{r}_2),\mathbf{Q},\mathtt{p},\mathtt{n}])=V([(+\infty,\mathtt{r}_2),\mathbf{Q},\mathtt{p},\mathtt{n}]),\\ \Rightarrow\ &\forall\mathtt{r}_1'\geq\mathtt{r}_1, V([(\mathtt{r}_1',\mathtt{r}_2),\mathbf{Q},\mathtt{p},\mathtt{n}]) = V([(+\infty,\mathtt{r}_2),\mathbf{Q},\mathtt{p},\mathtt{n}])\end{aligned}$$

Les ensembles d'états pour lesquels les valeurs sont égales forment sur la courbe de valeur correspondante des paliers. Nous retrouvons ces paliers sur la figure 6.2.

De plus, il existe un n-uplet de ressources maximal à consommer dans tous les sous espaces d'états $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$ d'une $\mathsf{PRU}_\mathtt{p}$ de la mission de taille P , que nous notons $\overline{\mathtt{r}^{\mathtt{max}}_{\mathbf{Q},\mathtt{p},\mathtt{n}}} = \sum_{\mathtt{p}=\mathtt{p}'}^{\mathsf{P}} \overline{\mathtt{r}^{\mathtt{max}}_{\mathtt{p}}}$. Il existe

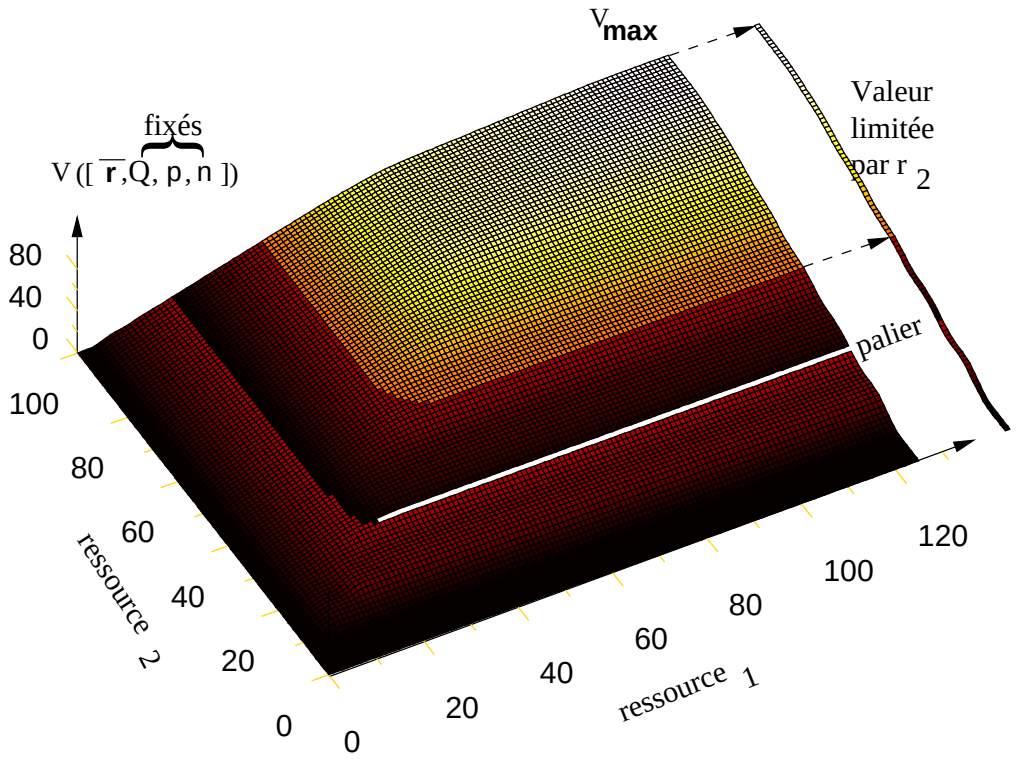


FIG. 6.2 – Paliers sur une fonction de valeur d'un sous espace d'états avec deux ressources.

Nous avons représenté sur la figure 6.2 la fonction de valeur pour un niveau $\mathtt{n}$ *d'une PRU* $\mathtt{p}$ *donnée à* $\mathbf{Q}$ *fixée. Chaque case représente un état, la quantité de ressources correspondantes étant indiquée en abscisse, et sa valeur en ordonnée. Les valeurs les plus élevées sont les plus claires. Nous observons l'existence de paliers (les lignes de valeurs constantes qui vont vers la droite ou vers le haut). Ces paliers sont dûs à une ressource limitante. Nous avons représenté sur la droite de ce graphique la fonction de valeur en une seule dimension en considérant que la ressource* $\mathtt{r}_1$ *n'intervenait plus dans la mission (on la considère donc comme infinie), donc que seule* $\mathtt{r}_2$ *varie.*

également une quantité de ressources maximale à consommer pour chaque type de ressources $\mathtt{r}^{\max}_{\mathbf{Q},\mathtt{p},\mathtt{n},1} = \sum_{\mathtt{p}=\mathtt{p}'}^{\mathtt{P}} \mathtt{r}^{\max}_{\mathtt{p},1}$ et $\mathtt{r}^{\max}_{\mathbf{Q},\mathtt{p},\mathtt{n},2} = \sum_{\mathtt{p}=\mathtt{p}'}^{\mathtt{P}} \mathtt{r}^{\max}_{\mathtt{p},2}$. En découle les deux égalités suivantes :

$$\begin{aligned} \forall \mathbf{Q},\mathtt{p},\mathtt{n},\forall \mathtt{r}_1 \in \mathtt{R}_1, V([(\mathtt{r}_1,+\infty),\mathbf{Q},\mathtt{p},\mathtt{n}]) &= V([(\mathtt{r}_1,\mathtt{r}^{\max}_{\mathbf{Q},\mathtt{p},\mathtt{n},2}),\mathbf{Q},\mathtt{p},\mathtt{n},]) \\ \forall \mathbf{Q},\mathtt{p},\mathtt{n},\forall \mathtt{r}_2 \in \mathtt{R}_2, V([(+\infty,\mathtt{r}_2),\mathbf{Q},\mathtt{p},\mathtt{n},]) &= V([(\mathtt{r}^{\max}_{\mathbf{Q},\mathtt{p},\mathtt{n},1},\mathtt{r}_2),\mathbf{Q},\mathtt{p},\mathtt{n},]) \end{aligned}$$

Notation : nous notons $V_1 : \mathtt{R}_1 \rightarrow \mathbb{R}$ telle que $\forall \mathtt{r}_1 \in \mathtt{R}_1, V_1(\mathtt{r}_1) = V([(\mathtt{r}_1,+\infty),\mathbf{Q},\mathtt{p},\mathtt{n}])$. De la même façon, nous généralisons cette notation à plus de deux ressources en plaçant en indice les ressources qui varient, les autres étant considérées comme infinies (ou maximales) :

$$\begin{aligned}
\forall \mathbf{Q}, \mathtt{p}, \mathtt{n}, \forall (\mathtt{r}_1, \ldots, \mathtt{r}_\omega, \ldots, \mathtt{r}_\Omega) \in \mathtt{R}, V_1([\mathtt{r}_1, \mathbf{Q}, \mathtt{p}, \mathtt{n}]) &= V([(\mathtt{r}_1, +\infty, \ldots, +\infty), \mathbf{Q}, \mathtt{p}, \mathtt{n}]) \\
\forall \mathbf{Q}, \mathtt{p}, \mathtt{n}, \forall (\mathtt{r}_1, \ldots, \mathtt{r}_\omega, \ldots, \mathtt{r}_\Omega) \in \mathtt{R}, V_{1,\omega}([(\mathtt{r}_1, \mathtt{r}_\omega), \mathbf{Q}, \mathtt{p}, \mathtt{n}]) &= V([(\mathtt{r}_1 \underbrace{, \ldots,}_{+\infty} \mathtt{r}_\omega, \underbrace{\ldots, +\infty}_{+\infty}), \mathbf{Q}, \mathtt{p}, \mathtt{n}])
\end{aligned}$$

Nous pouvons généraliser la propriété de ressource limitante pour deux ressources à trois ressources [41], et nous laissons le soin au lecteur de généraliser le principe à n ressources.

$$\forall (\mathtt{r}_1, \mathtt{r}_2, \mathtt{r}_3) \in \mathtt{R}, V([(\mathtt{r}_1, \mathtt{r}_2, \mathtt{r}_3)]) \leq \begin{cases} V_{1,2}([(\mathtt{r}_1, \mathtt{r}_2)]) \leq \begin{cases} V_1([(\mathtt{r}_1)]) \\ V_2([(\mathtt{r}_2)]) \end{cases} \\ V_{2,3}([(\mathtt{r}_2, \mathtt{r}_3)]) \leq \begin{cases} V_2([(\mathtt{r}_2)]) \\ V_3([(\mathtt{r}_3)]) \end{cases} \\ V_{1,3}([(\mathtt{r}_1, \mathtt{r}_3)]) \leq \begin{cases} V_1([(\mathtt{r}_1)]) \\ V_3([(\mathtt{r}_3)]) \end{cases} \end{cases}$$

Pour borner la fonction de valeur sur un espace d'états de dimension n, nous utilisons donc n fonctions de valeur sur des espaces d'états de dimension $n-1$, ces fonctions étant elles-mêmes bornées par d'autres fonctions de valeurs sur des espaces d'états de dimension $n-2$ etc... Au final, pour borner une fonction de valeur sur $\mathtt{R}$ il faut une fonction de valeur définie sur chaque élément de l'ensemble des parties de l'espace des ressources $\mathtt{R}$. Ces fonctions bornes sont calculées de manière indirecte si on utilise un algorithme simple. Nous allons les calculer puis les utiliser pour réduire le nombre de calculs.

6.3.2 Un algorithme pour accélérer le calcul de la fonction de valeur

L'accélération du calcul de la fonction de valeur pour une mission composée de PRUs prenant en compte plusieurs ressources se base sur plusieurs principes :

- utiliser les propriétés de la fonction de valeur et les bornes de l'espace d'états (celles qui ont été présentées plus haut),
- trouver un sens de parcours adéquat sur l'espace d'états afin de tirer parti de ces bornes,
- éviter d'évaluer certains états et éviter de calculer certains *« Bellman Backup »* [42].

Le sens de parcours adéquat

Pour accélérer le calcul de la fonction de valeur pour chaque état de la mission, nous avons implémenté un algorithme qui fonctionne niveau par niveau, comme dans le chapitre précédent, en utilisant :

[41] Nous considérons que $\mathbf{Q}, \mathtt{p}, \mathtt{n}$ sont fixés pour des raisons de lisibilité.

[42] Le *« Bellman Backup »* consiste à faire la somme de la valeur des états suivants possibles pondérée par les probabilités de transitions d'y arriver.

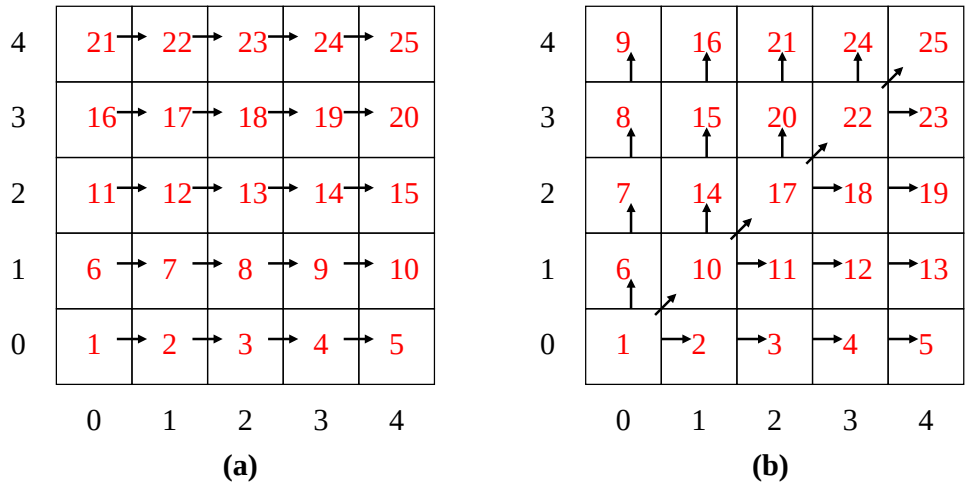


FIG. 6.3 – Parcours en diagonale d'un espace à deux dimensions.

Nous avons représenté sur la figure 6.3 deux parcours possibles d'un tableau à deux dimensions. Le sens de chaque parcours est indiqué par des flèches, et les étapes du parcours sont numérotées dans chaque case. Sur la figure 6.3.a, le sens de parcours est classique : on parcourt une ligne, puis la seconde, jusqu'à la dernière. Le parcours de droite (figure 6.3.b) est un parcours en diagonale : on commence par parcourir la première ligne, puis la première colonne, et on décale l'indice de parcours en diagonale (case 10*), puis on parcourt la deuxième ligne privée de son premier élément, la deuxième colonne privée de son premier élément etc...*

1. l'algorithme de création de clef niveau (algorithme 6),
2. l'algorithme d'évaluation de niveau (algorithme 7),
3. le tout englobé dans un parcours itératif inversé des PRUs de la mission (algorithme 8).

La première et la troisième étape ne dépendent pas du nombre de ressources à considérer, elles sont donc strictement identiques aux algorithmes du chapitre 5. Par contre, nous utilisons les remarques et les propriétés vues précédemment (concept de ressources limitantes) pour ne pas calculer les paliers existants dans la phase d'évaluation des niveaux. Pour cela, nous calculons la valeur de chaque état d'un sous espace $\mathcal{S}_{\mathbf{Q},\mathrm{p},\mathrm{n}}$ en parcourant cet espace selon sa diagonale, comme le schématise la figure 6.3. Ceci nous permet d'arrêter l'évaluation d'un ensemble d'états d'une même ligne (ou colonne) dès que le maximum est atteint. Ce principe est montré par l'algorithme 9 et la figure 6.4.

Borner chaque sous espace d'états

Mais avant d'appliquer cet algorithme, il faut connaître les maxima de valeurs du niveau dont nous sommes en train d'évaluer les états afin de pouvoir justement exploiter l'équation 6.3.1.0.

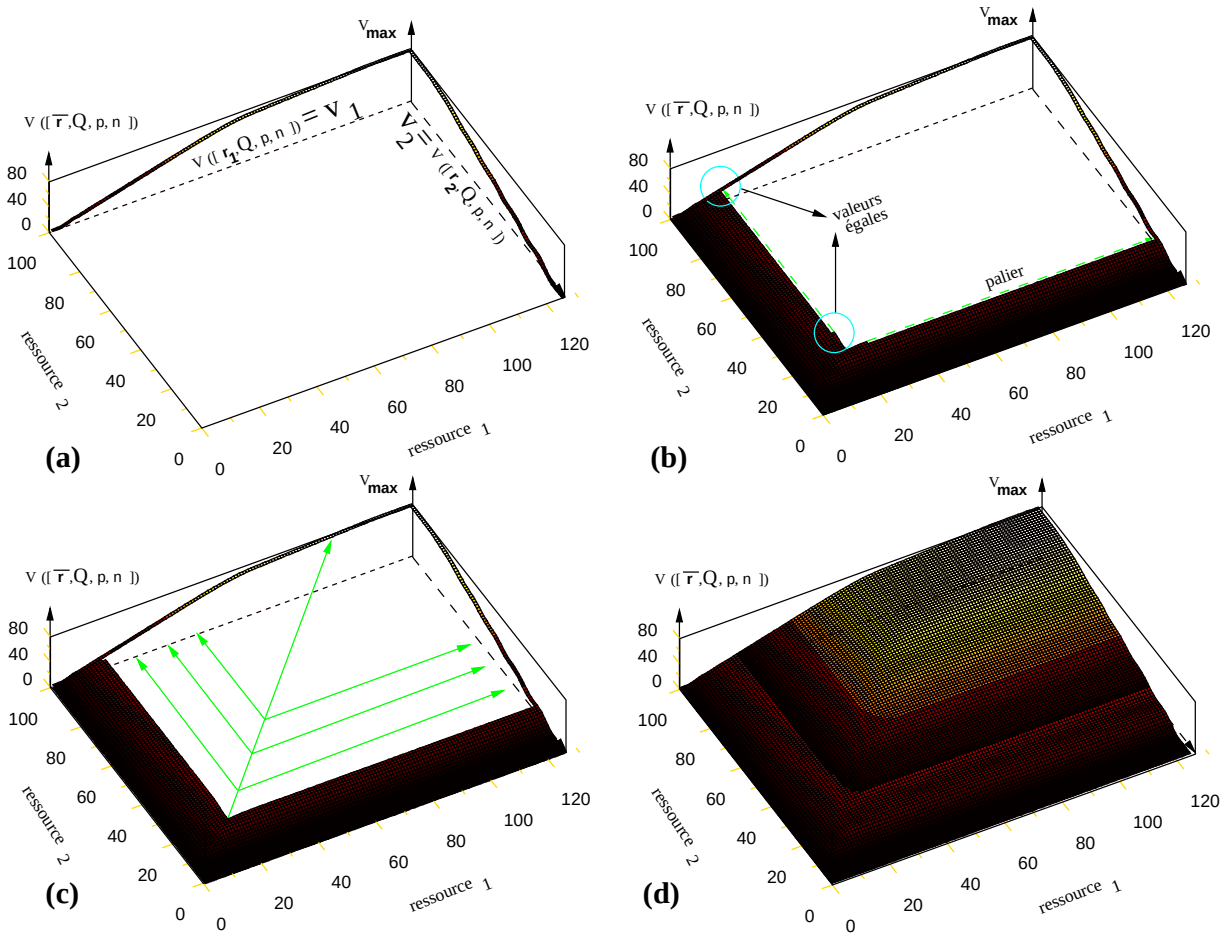


FIG. 6.4 – Construction de l'espace d'états.

Nous avons représenté sur la figure 6.4 les étapes de l'évaluation d'un sous espace d'états $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$. *Dans un premier temps (figure 6.4.a) nous bornons la fonction de valeur en calculant* $V_1(\mathtt{r_1}) = V([(\mathtt{r_1}, +\infty), \mathbf{Q}, \mathtt{p}, \mathtt{n}])$ *et* $V_2(\mathtt{r_2}) = V([(+\infty, \mathtt{r_2},), \mathbf{Q}, \mathtt{p}, \mathtt{n}])$. *Ensuite, on évalue les états un par un (figure 6.4.b) en suivant le sens de parcours indiqué sur la figure 6.3. Dès que la valeur d'un état est égale à celle de* V_1 *ou de* V_2 *(selon le sens du parcours suivi) on stoppe l'évaluation pour générer directement les valeurs des états du palier (figure 6.4.c). Finalement, on continue à « monter » en suivant le sens de parcours diagonal (figure 6.4.d).*

Nous notons tout simplement $V_1 : \mathtt{R_1} \rightarrow \mathbb{R}$ telle que $\forall \mathtt{r_1} \in \mathtt{R_1}, V_1(\mathtt{r_1}) = V([(\mathtt{r_1}, +\infty), \mathbf{Q}, \mathtt{p}, \mathtt{n}])$ (même chose pour V_2) et nous opérons préalablement une évaluation de ces deux fonctions mono-ressources comme dans le chapitre précédent, niveau par niveau, en considérant tout simplement que la ressource infinie ne diminue[43] pas. Ceci correspond à l'étape (**a**) de la figure 6.4.

Les fonctions V_1 et V_2 nous fournissent alors deux bornes pour les valeurs des états, et nous

[43] Nous considérons même qu'elle n'existe pas.

plaçons les fonctions correspondantes respectivement en haut et à droite de l'espace d'états. C'est seulement après avoir évalué ces bornes que nous commençons l'algorithme 9.

Sur un palier donné, au lieu de calculer la valeur de chaque état en appliquant le *« Bellman Backup »* , nous générons ces états en leur affectant directement la bonne valeur. Cette valeur est connue, c'est celle qui à été calculée dans la fonction borne V_1 (ou V_2). En évitant de calculer le *« Bellman Backup »* , l'accélération est nette, et les résultats sont dans le chapitre 10.

```
1  i_diag = 0
2  j_diag = 0
3  tant que i_diag ≤ r_1^max et j_diag ≤ r_2^max faire
4      pour i dans [i_diag, r_1^max] faire
5          v = evalue([(i, j_diag), Q, p, n]) //evalue est le max de la somme pondérée des
           valeurs des états suivants si v == S[(r_1^max, j_diag), Q, p, n].valeur alors
6              //remplissage de la ligne avec la même valeur v
7              pour i' dans [i, r_1^max] faire
8                  S[(i', j_diag), Q, p, n].valeur = v
9              finpour
10             break
11         finsi
12     finpour
13     pour j dans [j_diag, r_2^max] faire
14         v = evalue([(i_diag, j), Q, p, n])
15         si v == S[(i_diag, r_2^max), Q, p, n].valeur alors
16             //remplissage de la colonne avec la même valeur v
17             pour j' dans [j, r_2^max] faire
18                 S[(i_diag, j'), Q, p, n].valeur = v
19             finpour
20             break
21         finsi
22     finpour
23     i_diag ++
24     j_diag ++
25 fintq
```

Algorithme 9 : Calculer les valeurs des états d'un même niveau $\mathtt{n}$, à $\mathbf{Q}$ fixé

Note : la fonction **evalue** de cet algorithme calcule la valeur de l'état courant (**evalue**(**s**) =

$V(\mathbf{s}) = \sum_{\mathbf{s}'} \dots$) en fonction des états suivants , dont la valeur est connue à ce stade. C'est cette fonction que nous évitons en partie avec l'algorithme 9. Dans ce qui suit, nous ne générons même pas les états là où **evalue** n'est pas calculée, pour gagner de l'espace mémoire. Les états « paliers » seront agrégés en un seul.

6.3.3 Calcul de la fonction de valeur avec agrégation de l'espace d'états

L'algorithme de calcul de la fonction de valeur avec agrégation de l'espace d'états fonctionne comme l'algorithme 9 à ceci prêt que nous allons regrouper certains états de chaque sous espace d'états $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$ qui ont la même valeur, de façon à réduire l'espace mémoire utilisé pendant ce calcul. Cette agrégation est exacte et non uniforme. La fonction de valeur obtenue est la fonction de valeur optimale, nous voulons un résultat sans approximation. C'est seulement la représentation de l'espace d'état sur laquelle V est définie qui est adaptée au contexte des ressources multiples.

Le calcul de la fonction de valeur par le mécanisme de programmation dynamique demande de stocker chaque état et sa valeur, et l'algorithme fera un ou plusieurs accès à la valeur de ces états à chaque évaluation, par le biais du *« Bellman Backup »* . Il va donc falloir trouver un compromis entre la compression obtenue par la représentation agrégée et le temps nécessaire pour accéder aux données, qui pourrait ralentir le calcul. Il faut trouver une structure qui permette à la fois :

- de diminuer la représentation de l'espace d'états,
- d'avoir la possibilité de retrouver rapidement la valeur d'un état.

Nous proposons deux structures pour l'agrégation, la première technique d'agrégation va générer un espace d'états moins compact que la deuxième, mais permettre un accès plus rapide.

Une agrégation faible

Nous allons agréger chaque sous espace d'états $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$ pour réduire la représentation de la fonction de valeur pendant le calcul. Cette compression passe par une agrégation de certains états qui ont la même valeur, elle est donc exacte. Ces états à regrouper sont ceux des paliers que nous avons présentés dans la section précédente. Les techniques utilisées sont similaires à celles que nous venons de présenter pour l'accélération du calcul de la politique.

Nous utilisons une structure pour représenter l'espace d'états similaire à celle du sens de parcours diagonale. En deux dimensions, nous avons dans cette structure :

- une liste qui représente la diagonale de l'espace à représenter, chaque élément i de cette liste diagonale est un triplet (valeur, ligne, colonne) où :
  - ligne est une liste d'états dont l'élément j représente la valeur de l'état $[(\mathtt{j},\mathtt{i}),\mathbf{Q},\mathtt{p},\mathtt{n}]$, j variant de $i+1$ à ${\mathtt{r}_1}^{max}-1$,

- colonne est une liste d'états dont l'élément j représente la valeur de l'état $[(\mathtt{i},\mathtt{j}),\mathbf{Q},\mathtt{p},\mathtt{n}]$, j variant de $i+1$ à ${\mathtt{r}_2}^{max}-1$,
- valeur est la valeur de l'état $[(\mathtt{i},\mathtt{i}),\mathbf{Q},\mathtt{p},\mathtt{n}]$,

– la fonction borne V_1 où les ressources $\mathtt{r}_2$ sont présentes en quantité illimitées,

– la fonction borne V_2 où les ressources $\mathtt{r}_1$ sont présentes en quantité illimitées.

La structure que nous venons de présenter permet d'énumérer tous les états d'un sous espace d'états donné $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$. Nous allons appliquer une agrégation sur cette structure : chaque palier (voir figure 6.2) de fin de ligne ou de colonne va être codé comme un seul état, et tous les autres états, qui se situent avant ce palier vont être stockés respectivement dans les listes ligne$[i]$ et les listes colonne$[i]$. Cette structure agrégée est représentée sur la figure 6.5. Chaque sous liste ligne$[i]$ contiendra les états $[(\mathtt{j},\mathtt{i}),\mathbf{Q},\mathtt{p},\mathtt{n}]$, tels que $j > i$ et $V([(\mathtt{j},\mathtt{i}),\mathbf{Q},\mathtt{p},\mathtt{n}]) < V_2(i)$. Chaque sous liste colonne$[i]$ représentera au contraire les états dont la valeur est inférieure à $< V_1(i)$. Formellement à $\mathbf{Q},\mathtt{p},\mathtt{n}$, fixés :

$$\text{ligne}[i] = \{[(\mathtt{j},\mathtt{i}),\mathbf{Q},\mathtt{p},\mathtt{n}],\ j > i \text{ et } V([(\mathtt{j},\mathtt{i}),\mathbf{Q},\mathtt{p},\mathtt{n}]) < V_2(i)\} \tag{6.12}$$

$$\text{colonne}[i] = \{[(\mathtt{i},\mathtt{j}),\mathbf{Q},\mathtt{p},\mathtt{n}],\ j > i \text{ et } V([(\mathtt{i},\mathtt{j}),\mathbf{Q},\mathtt{p},\mathtt{n}]) < V_1(i)\} \tag{6.13}$$

Nous appelons cette agrégation *faible* parce que sur chaque sous liste de la diagonale, il n'y a qu'un seul état qui est agrégé, le dernier.

Pour retrouver la valeur d'un état dans un sous espace d'états nous avons une complexité en temps constant $\mathcal{O}(1)$. Le sens de parcours de notre structure pour deux ressources est le suivant. Étant donné un n-uplet de ressources $\overline{\mathtt{r}} = (\mathtt{r}_1,\mathtt{r}_2)$ on remonte dans la liste diagonale jusqu'au $\min(\mathtt{r}_1,\mathtt{r}_2)$, ensuite on va choisir dans la liste correspondante la ressource qui est en excès le numéro de cette ressource étant donnée par l'argmax suivant : $\text{argmax}_{1,2}(\mathtt{r}_1,\mathtt{r}_2)$. Dans cette sous-liste (ligne si 1 ou colonne si 2) on choisit enfin l'élément $|\mathtt{r}_2 - \mathtt{r}_1|$. Si cet élément est dans la liste, on le retourne (accès constant) sinon, on retourne la valeur qui se situe dans l'une des fonctions V_1 et V_2 qui bornent la fonction de valeur (accès constant également).

```
1 essayer :
2 retourner listeDiagonale[min(x,y)][argmax(x,y)][max(x,y)]
3 sinon :
4 retourner bornes[argmax(x,y)][min(x,y)]
```

Algorithme 10 : Retrouver la valeur d'un état dans un espace agrégé en temps constant.

Si par exemple pour un sous espace d'états donné (donc à $\mathbf{Q},\mathtt{p},\mathtt{n}$ fixé) comme celui de la figure 6.5, nous voulons retrouver la valeur de l'état $(\mathtt{r}_1,\mathtt{r}_2) = (3,2)$, on accède à la case 2 de la liste diagonale, le min de $(2,3)$. Ensuite, comme il y a plus de ressources $\mathtt{r}_1$ que de $\mathtt{r}_2$, on se place

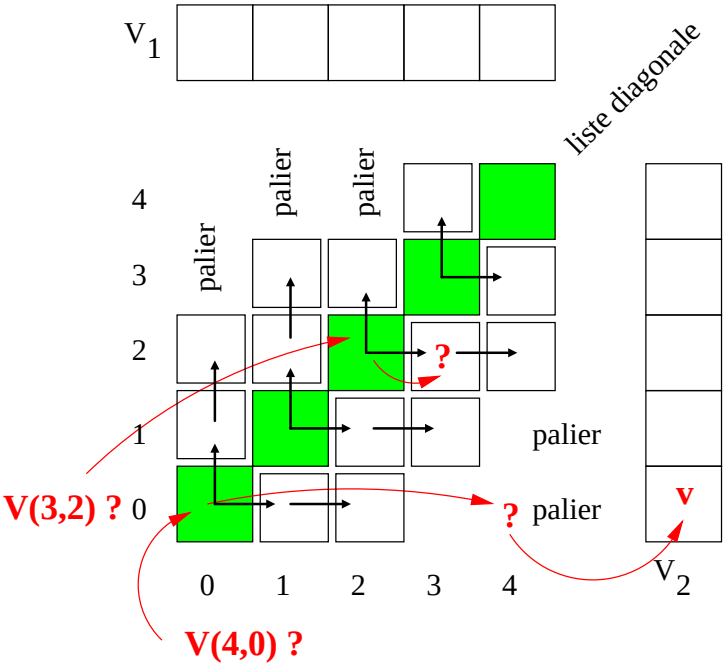


FIG. 6.5 – Structure agrégée représentant un sous espace d'états.

La figure 6.5 illustre la structure utilisée pour agréger chaque sous espace d'états. Le cœur de la structure est composée d'une liste diagonale, et de deux fonctions bornes V_1 et V_2 qui sont les fonctions représentant la valeur d'un état où la ressource $\mathtt{r}_2$ et respectivement $\mathtt{r}_1$ serait présente en quantité maximale (ou illimitée). La liste diagonale est une liste composée à chaque nœud de deux sous listes allant vers le haut (colonne) pour représenter les couples où $\mathtt{r}_1 < \mathtt{r}_2$ et vers la droite (ligne) pour l'inverse. Les états paliers, c'est-à-dire ceux dont la valeur est égale à une des valeurs de V_1 et V_2 ne sont pas stockés dans les listes lignes et colonnes, on ira chercher leur valeur dans les fonctions bornes V_1 et V_2.

dans la sous liste qui part sur la droite. On accède à la première valeur de cette sous liste puisque $3 - 2 = 1$. Pour retrouver la valeur d'un état non stocké, c'est-à-dire un état faisant partie d'un palier, comme par exemple $(\mathtt{r}_1, \mathtt{r}_2) = (4, 0)$, on part du zéroième élément la liste diagonale et on cherche à accéder au quatrième élément de la sous liste droite. Comme cet élément n'existe pas, on accède directement à la valeur stockée dans la sous liste V_2 en 0.

L'algorithme qui permet de calculer la fonction de valeur pour une mission donnée est strictement identique à ceux que nous avons présentés dans la section précédente, à ceci près que l'algorithme 9 est modifié : nous ne générons pas les états du palier avec une boucle itérative, nous supprimons donc les lignes 7,8 et 9, et les lignes 16,17 et 18. L'instruction **break** permet de ne pas générer les états du palier.

Cette agrégation a donné de bons résultats tant au niveau de l'espace mémoire utilisé que

du temps de calcul, les performances sont données dans le chapitre 10. Cependant, nous avons essayé de faire mieux, d'agréger encore plus chaque sous espace d'états, pour gagner en espace mémoire, au risque de ralentir l'algorithme de calcul.

Une agrégation forte de l'espace d'états

Comme précédemment, nous agrégeons les états sur chaque sous espace $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$. L'agrégation est plus forte dans le sens où tous les états contigus ayant la même valeur sont représentés par un seul état agrégé. Ce type d'**agrégation** est aussi **exacte** et **non uniforme** (voir chapitre 3). Nous allons repérer les paliers dûs aux ressources limitantes comme dans la section précédente, mais aussi les paliers dus à des modules dont les ressources consommées sont éloignées. Si le domaine de consommation de ressources d'un module $\mathtt{a}$ ($\mathsf{m}_{\mathtt{p},\mathtt{n},a}$) est éloigné du domaine de consommation de ressources d'un module $\mathtt{b}$, ($\mathsf{m}_{\mathtt{p},\mathtt{n},b}$), alors aucun changement de valeur ne s'opérera entre ces deux domaines de valeurs. Ces types de paliers étaient déjà visibles avec une seule ressource nous en donnons un exemple dans la figure 6.6. Le but est de représenter une seule fois chaque valeur de chaque sous espace d'états, quand c'est possible.

C'est en tenant compte de ces deux types de paliers que nous obtenons une agrégation de chaque sous espace d'états $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$ comme le montre la figure 6.7.

Le principe de l'agrégation forte est le suivant : dans un sous espace d'états donné, si deux états ont la même valeur, nous les regroupons dans un état agrégé. Pour pouvoir retrouver ultérieurement la valeur d'un état pour un montant de ressource donné dans un sous espace agrégé de cette façon, nous conservons les états clefs qui forment la frontière inférieure de chaque état agrégé. Ces états clefs sont calculés en utilisant la relation d'ordre partiel $\prec$ sur $\mathtt{R}$. Un état agrégé est décrit par l'ensemble des états clefs qui forment la frontière inférieure minimale (au sens des ressources) pour une valeur donnée.

Le principe d'agrégation est illustré sur la figure 6.7. L'espace d'états agrégé est constitué d'une liste des différentes valeurs que peut prendre la fonction V sur ce sous espace, et dans chaque élément de la liste, qui est un état agrégé, nous conservons la partie minimale des états qui forment la borne inférieure de cet état : les états clefs (voir figure 6.7.c)

Formalisons cette partie minimale : dans un sous espace d'états $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$, nous regroupons tout d'abord tous les états ayant une valeur égale v : $\mathcal{S}_v = \{\mathbf{s} \in \mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}, V(\mathbf{s}) = v\}$. On montre facilement que cette partition recouvre tout le sous espace d'états $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$.

$$\bigcup_{v\in[0,v_{\max}]} \mathcal{S}_v = \mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}} \tag{6.14}$$

$$\forall v, v' \in [0, v_{\max}]^2, \mathcal{S}_v \cap \mathcal{S}_{v'} = \emptyset$$

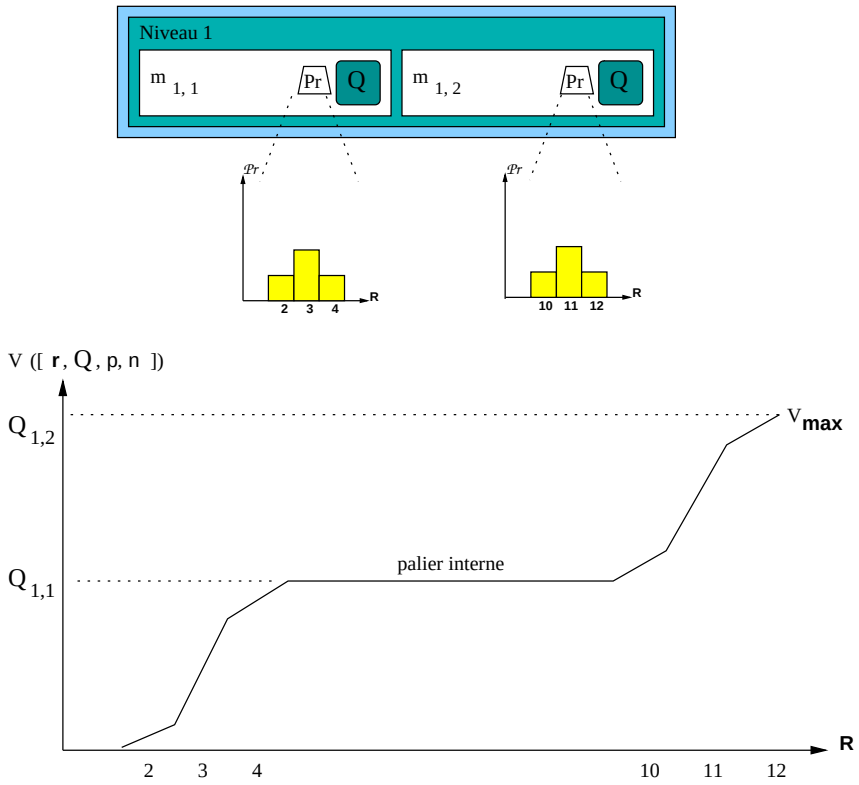


FIG. 6.6 – Palier sur une fonction de valeur avec une seule ressource.

Explications *pour la figure 6.6 : soit une PRU avec un seul niveau, et deux modules. Le premier module produit une qualité* $\mathbf{Q}_{1,1} \leq \mathbf{Q}_{1,2}$. *Comme le module de droite consomme au moins* 10 *ressources, les états dans lesquels il reste de* 4 *à* 9 *ressources ont des valeurs identiques. On obtient un palier.*

Pour une valeur possible v nous définissons ϕ_v comme une partie des états $\mathcal{S}_v$, tel que pour tout état de $\mathcal{S}_v$, il existe au moins un état dans ϕ_v dont le montant de ressources est inférieur ou égal au sens de la relation de dominance (voir équation 12). Une partie ϕ_v est, en quelques sortes, une frontière inférieure de $\mathcal{S}_v$. Formellement une ϕ_v est définie par la relation suivante :

$$\forall \mathbf{s}' = [\overline{\mathbf{r}}', \mathbf{Q}, \mathbf{p}, \mathbf{n}] \in \mathcal{S}_v, \exists \mathbf{s} = [\overline{\mathbf{r}}, \mathbf{Q}, \mathbf{p}, \mathbf{n}] \in \phi_v \quad \text{tel que} \quad \overline{\mathbf{r}} \preceq \overline{\mathbf{r}}' \tag{6.15}$$

Nous notons $\mathcal{P}_v$ l'ensemble des parties de $\mathcal{S}_v$ qui sont des ϕ_v ; la partie minimale $\phi_v^{\min}$ est celle où il y a le moins d'états clefs. Elle est définie par la relation suivante :

$$\phi_v^{\min} = \arg\min_{\phi_v \in \mathcal{P}_v} |\phi_v|. \tag{6.16}$$

Finalement, la représentation de l'espace d'états agrégé se limite à un liste de $\phi_v^{\min}$:

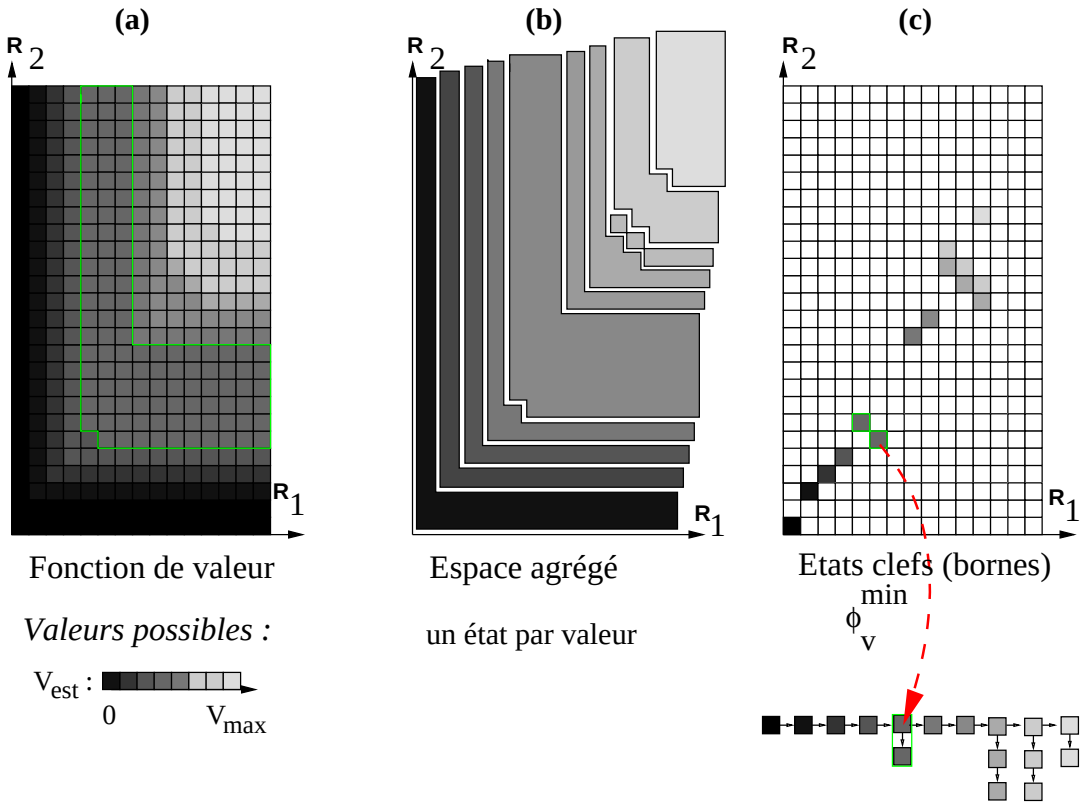


FIG. 6.7 – Agrégation forte d'un sous espace d'états $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$.

Explications *pour la figure 6.7. Les ressources sont disposées selon les deux axes vertical et horizontal. La figure 6.7.a représente la fonction de valeur pour un sous espace d'états* $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$ *donné , chaque case représente un montant de ressources, la valeur de chaque état est représentée par un niveau de gris, le blanc représentant un maximum de valeur, le noir le* 0*. Les états qui ont la même valeur sont agrégés sur la figure 6.7.b. La figure 6.7.c est une représentation des états clefs* ($\phi_v^{\min}$) *qui bornent la partie inférieure (au sens de la ressource) des états agrégés.*

$$\mathcal{S}^{ag}_{\mathbf{Q},\mathtt{p},\mathtt{n}} = \bigcup_{v\in[\,0,v_{\max}]} \phi_v^{\min} \tag{6.17}$$

Nous avons implémenté cette agrégation (voir l'algorithme 11).

Une fois l'espace agrégé, il faut pouvoir retrouver la valeur d'un état $\mathbf{s} = [\overline{\mathtt{r}}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]$ en utilisant cette nouvelle représentation. Formellement pour un montant de ressources donné, la valeur de l'état correspondant est donnée par :

$$V([\overline{\mathtt{r}}, \mathbf{Q}, \mathtt{p}, \mathtt{n}]) = \max_v\{v, \exists[\overline{\mathtt{r}}', \mathbf{Q}, \mathtt{p}, \mathtt{n}] \in \phi_v^{\min},\ \text{tel que } \overline{\mathtt{r}}' \preceq \overline{\mathtt{r}}\} \tag{6.18}$$

Autrement dit, la valeur d'un état est la valeur maximale parmi les états agrégés pour lesquels

Données : Les états évalués des niveaux suivants
Résultat : $\mathcal{S}_{ag}$

$\mathcal{S}_{ag} = \emptyset$ //dictionnaire vide de listes de couples de ressources
pour $(\mathtt{r}_1, \mathtt{r}_2) \in \mathtt{R}$ **faire**
  //Le sens de parcours est diagonal
  $v = \mathcal{S}[(\mathtt{r}_1, \mathtt{r}_2), \mathbf{Q}, \mathtt{p}, \mathtt{n}]$.valeur
  si $\mathcal{S}_{ag}[v] == \emptyset$ **alors**
    //S'il n'y a pas d'indice de valeur v dans l'espace agrégé
    $(\phi_v^{\min}) =$ **new**$(\phi^{\min})$ //On crée un nouveau sous-ensemble (dictionnaire)
    $\mathcal{S}_{ag}[v] = (\phi_v^{\min})$
  finsi
  si $\phi_v^{\min} == \emptyset$ **alors**
    $\phi_v^{\min}$.**ajoute**$((\mathtt{r}_1, \mathtt{r}_2))$
  finsi
  $ajoutNecessaire = VRAI$
  pour $(\mathtt{r}'_1, \mathtt{r}'_2) \in \phi_v^{\min}$ **faire**
    si $(r'_1 \leq r_1)$***et***$(r'_2 \leq r_2)$ **alors**
      $ajoutNecessaire = FAUX$
    finsi
    si $ajoutNecessaire$ **alors**
      $\phi_v^{\min}$.**ajoute**$((\mathtt{r}_1, \mathtt{r}_2))$
    finsi
  finpour
finpour
retourner $\mathcal{S}_{ag}$

Algorithme 11 : Agrégation forte d'un sous espace d'états $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$

il existe au moins un état clef dont la quantité de ressources est inférieure ou égale à celle de l'état considéré.

Pour retrouver la valeur d'un état donné (avec une certaine quantité de ressources), nous parcourons les $\phi_v^{\min}$ dans le sens croissant de leur valeur, et la valeur d'un état est égale à la valeur du dernier état agrégé rencontré pour lequel il existe au moins un état $\mathbf{s}'$ dans $\phi_v^{\min}$ tel que $\overline{\mathtt{r}}' \preceq \overline{\mathtt{r}}$. Si le groupement d'états suivant ne contient pas d'état dont la quantité de ressources est inférieure ou égale, l'algorithme s'arrête. L'algorithme 12 permet de retrouver la valeur d'un état.

La complexité de l'algorithme permettant de retrouver la valeur pour un montant de res-

Données : un état $\mathbf{s} = ([\overline{\mathtt{r}}, \mathbf{Q}, \mathtt{p}, \mathtt{n}])$, un espace d'états agrégé $\mathcal{S}_{ag}(\mathbf{Q}, \mathtt{p}, \mathtt{n})$
Résultat : $V([\overline{\mathtt{r}}, \mathbf{Q}, \mathtt{p}, \mathtt{n}])$

```
1 V = 0
2 i = 1
3 tant que i ≤ longueur(S_ag) faire
4 |   si ∃r̄' ∈ S_ag[i], r̄' ≺ r̄ alors
5 |   |   V = V(S_ag[i])
6 |   finsi
7 |   i++
8 fintq
9 retourner V([r̄, Q, p, n])
```

Algorithme 12 : Retrouver la valeur d'un état

sources donné est supérieure à un temps constant, la complexité de l'algorithme d'agrégation forte également (pour insérer un état donné). De ce fait, le calcul de la fonction de valeur en utilisant cette agrégation forte est plus lent que la méthode de la section précédente.

Cependant l'espace agrégé fortement demande moins de valeurs à stocker qu'avec l'agrégation faible. C'est un complément de la méthode d'agrégation faible pour pouvoir stocker la fonction de valeur sur l'espace d'états $\mathcal{S}_{\mathbf{0},\mathtt{p},0}$, après que le calcul ait été effectué en la représentant sur un espace le plus petit possible. En effet, nous avons vu que la fonction de valeur servirait à recalculer la politique de la PRU $\mathtt{p}$ courante. Par conséquent, nous aurons besoin de connaître la valeur de **tous** les états du sous espace d'états $\mathcal{S}_{\mathtt{p},0,\mathbf{0}}$.

C'est pour cela que nous avons implémenté un algorithme qui permet de retrouver toutes ces valeurs en une fois plutôt qu'une seule valeur à chaque fois. En répétant plusieurs fois l'algorithme 12, nous obtenons des résultats catastrophiques puisqu'il faut réutiliser plusieurs fois le parcours de la liste structurée, dont la complexité dans le pire des cas est égale aux nombres de valeurs différentes que peuvent prendre les états.

Avec l'algorithme dont le principe est en figure 6.8, nous retrouvons toutes les valeurs en une seule passe sur le sous espace d'états. Le principe de cet algorithme est le suivant : pour une clef $\mathbf{Q}, \mathtt{p}, \mathtt{n}$ donnée, nous générons le sous espace d'états $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$ correspondant, dont la taille est égale au n-uplet ressources maximal $\overline{\mathtt{r}_{\max}}$. Nous utilisons un parcours diagonal pour évaluer chaque état, en commençant donc par la case $\overline{0} = (0, \ldots, 0)$. La valeur de cette première case est connue puisque cet état fait partie des états stockés dans notre structure compressée représentée sur la figure 6.7.b. La valeur d'un état est ensuite égale au maximum de la valeur des états qui le précèdent d'une case (sur la partie droite de la figure 6.8, les états qui précèdent l'état 12 sont, en deux dimensions, celui de gauche : X, et celui en dessous : 11). Formellement, à $\mathbf{Q}, \mathtt{p}, \mathtt{n}$, fixés :

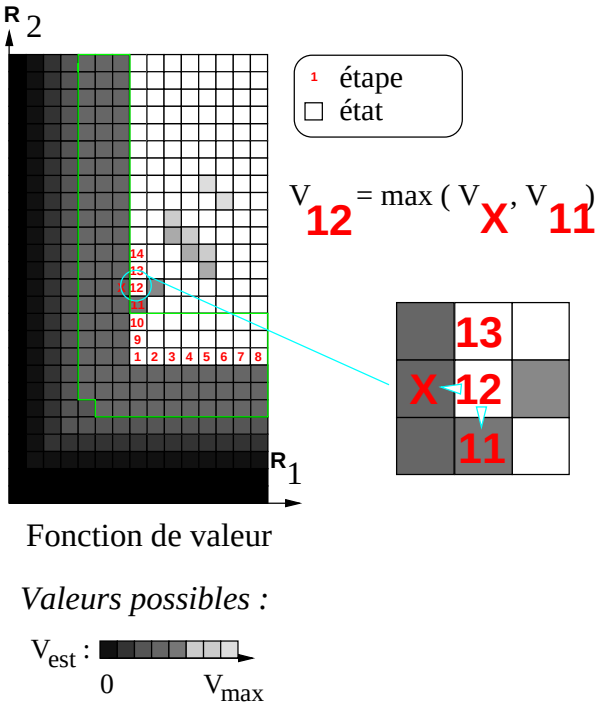


FIG. 6.8 – Reconstruction d'un sous espace d'états fortement agrégé.

Nous expliquons sur la figure 6.8 comment retrouver la valeur de ***tous*** *les états d'un sous espace d'états* $\mathcal{S}_{\mathbf{Q},\mathtt{p},\mathtt{n}}$. *Chaque état est représenté par une case, sa valeur est d'autant plus claire qu'elle est grande. L'algorithme utilise un sens de parcours diagonal (comme en 6.3, les états sont numérotés dans chaque case. Les états grisés ont une valeur déjà connue : leur valeur est enregistrée dans la structure agrégée de la figure 6.7.b. La valeur des états grisés a déjà été calculée puisque ce sont des états clefs appartenant aux* $\phi_v^{\min}$. *La valeur de l'état qui est représenté par la case* 1 *est égale à la valeur maximale entre la valeur de l'état grisé à sa gauche et en dessous, même chose pour la case* 2 *etc... Pour la case* 11 *par contre, il suffit de garder la valeur de cet état qui est déjà connue. Pour la case* 12, *la valeur est égale au maximum entre les deux états justes inférieurs qui le jouxtent. Ici* $V(11) > V(X)$, *donc* $V(12) = V(11)$.

$$V(\mathtt{r}_1,\ldots,\mathtt{r}_\omega,\ldots,\mathtt{r}_\Omega) \;=\; \max_{\omega\in\{1..\Omega\}} \begin{pmatrix} V(\mathtt{r}_1-1,\ldots,\mathtt{r}_\omega,\ldots,\mathtt{r}_\Omega) \\ \ldots \\ V(\mathtt{r}_1,\ldots,\mathtt{r}_\omega-1,\ldots,\mathtt{r}_\Omega) \\ \ldots \\ V(\mathtt{r}_1,\ldots,\mathtt{r}_\omega,\ldots,\mathtt{r}_\Omega-1) \end{pmatrix} \tag{6.19}$$

Nous continuons ensuite de suivre ce parcours diagonal jusqu'à ce que $\overline{\mathtt{r}_{\max}}$ soit évalué. Cette méthode permet de retrouver la valeur de tous les états d'un sous espace d'états donné avec une

complexité égale à la taille de ce sous espace d'états, cette algorithme retrouve en temps linéaire la valeur de chaque état.

Finalement, nous ne pouvons pas utiliser cette méthode pour faire de l'agrégation pendant le calcul de la fonction de valeur, puisque l'algorithme qui permet de retrouver la valeur d'un état seul est de complexité en $\mathcal{O}(\Omega)$. Si nous utilisons l'algorithme de programmation dynamique par niveau avec la méthode d'agrégation forte, il va falloir accéder plusieurs fois à la valeur de chaque état de $\mathcal{S}_{\mathbf{Q},\mathrm{p},\mathrm{n}}$ pendant le calcul de la valeur des états de $\mathcal{S}_{\mathbf{Q}',\mathrm{p},\mathrm{n}-1}$ via le Bellman Backup. Si nous choisissons de recomposer $\mathcal{S}_{\mathbf{Q},\mathrm{p},\mathrm{n}}$ entièrement avant le calcul de $\mathcal{S}_{\mathbf{Q}',\mathrm{p},\mathrm{n}-1}$, l'agrégation ne sert à rien.

Choix de la méthode d'agrégation

Finalement, lors du calcul de la fonction de valeur, nous utilisons la méthode d'agrégation faible pour chaque sous espace d'états, qui nous permet d'accélérer le calcul tout en limitant l'explosion de la taille de l'espace d'états. Les résultats sont dans le chapitre 10.

Conclusion

Nous avons adapté le raisonnement progressif aux ressources multiples. Notre adaptation s'est faite en deux temps. Dans un premier temps, nous avons modifié le modèle de description des PRUs de façon à inclure une consommation incertaine de ressources multiples dans chaque module, en modifiant les fonctions de distribution de probabilités.

La programmation dynamique appliquée telle quelle, comme dans le chapitre précédent, ne suffit plus pour calculer une politique dans un temps raisonnable pour une mission de quelques PRUs, l'espace d'états explose avec l'ajout de nouvelles dimensions, ce problème est plus connu sous le nom de *Curse of dimentionnality*. Nous avons donc proposé un algorithme d'évaluation des états de la mission qui exploite la propriété de ressource limitante pour éviter de calculer la valeur de certains états qui en les regroupant par paliers. Nous proposons un algorithme d'agrégation faible qui agrège chaque palier en un seul état, permettant ainsi de réduire l'espace d'états, et de limiter le nombre de calculs (*Bellman Backup*) effectués. Des expériences montreront l'efficacité de cet algorithme dans le chapitre 10.

Aussi dans un troisième temps, nous avons adapté la structure de représentation des états, en nous appuyant sur la technique d'agrégation par valeurs. Nous ne gardons cette fois qu'un seul état par valeur, et nous le représentons par un ensemble minimal d'états clés. Cette agrégation n'est pas utilisable pour le calcul de la fonction de valeur, mais permet de réduire la représentation de la fonction de valeur sur laquelle le robot se base pour calculer sa politique locale.

Toutes ces améliorations nous permettent d'envisager le passage aux ressources multiples et donc de modéliser plus fidèlement les ressources que notre robot sera sensé réellement consommer.

Mais, même si l'algorithme que nous proposons accélère le temps de calcul de la fonction de valeur de l'ensemble des états, ce temps est trop long pour pouvoir envisager de laisser le robot calculer la fonction de valeur (en entier) pendant la mission. Nous devons donc nous limiter pour l'instant à des missions « statiques » dès que de multiples ressources entrent en compte, c'est-à-dire calculer la fonction de valeur grâce à laquelle l'agent trouvera sa politique optimale avant d'exécuter la mission. A ce stade, la gestion des imprévus dans une mission avec des ressources multiples n'est pas encore possible, puisque ceci demande de recalculer la fonction de valeur en entier dès qu'un changement intervient dans cette mission.

Conclusion de la deuxième partie

Afin de pouvoir confier certaines missions à un robot autonome dans des endroits inaccessibles, nous voulons fournir à ce robot un mécanisme de contrôle qui lui permette de gérer au mieux les ressources consommables embarquées : son énergie, son espace mémoire etc...

Nous voulions un modèle qui permette un contrôle flexible : pouvoir s'arrêter pendant une tâche donnée, et disposer de plusieurs alternatives pour effectuer les tâches. Le raisonnement progressif que nous avons présenté dans le chapitre 4 permet de modéliser un ensemble de tâches complexes, de gérer l'incertitude des ressources consommées pendant la mission, et de fournir un mécanisme de contrôle inspiré de celui qui existe pour les algorithmes anytimes : interruptibilité, qualité croissante de la solution.

Après avoir formalisé dans le chapitre 5 le raisonnement progressif, nous avons vite vu le principal problème lié à ce modèle qui permet de gérer l'incertitude des ressources consommées pendant la mission : l'espace d'états généré augmente au carré avec le nombre de PRUs présentes dans la mission et exponentiellement avec le nombre de ressources différentes. De ce fait, le temps nécessaire pour calculer la fonction de valeur et la politique associée devient grand. Notons tout de même que la complexité des algorithmes que nous présentons pour évaluer l'ensemble des états est proportionnel au nombre d'états, puisqu'ils ne sont évalués qu'une seule fois.

Le début de solution proposé par A.I. Mouaddib et S. Zilberstein est basé sur le principe de *"diviser pour régner"*. Grâce à la technique du coût occasionné qui mesure la différence entre ce qui peut être gagné localement et ce qui aurait pu être gagné dans le futur, nous avons un contrôle local de la consommation de ressources. Ceci permet donc de séparer le calcul entre :

- le présent, la tâche courante, pour laquelle l'agent calcule une politique ;
- le futur pour lequel l'agent calcule le gain espéré en fonction des ressources restantes.

Les deux sont, bien entendu, liés puisque la politique calculée localement dépend du gain espéré pour le reste de la mission.

Ceci permet, dans un premier temps, de limiter les calcul à la tâche courante (à chaque fois que l'agent va changer de tâche, son module de planification recalcule la politique correspondante en fonction du gain espéré en quelques millisecondes). Néanmoins, il faut garder la fonction de gain espéré pour le reste de la mission en mémoire, ou alors, trouver un moyen de la calculer

rapidement, ce qui n'est pour l'instant pas possible.

Finalement, nous apportons une première extension du raisonnement progressif : la prise en compte de multiples ressources. L'extension du modèle que nous adoptons est la suivante : nous remplaçons la ressource simple $\mathtt{r}$ par une ressource multidimensionnelle $\overline{\mathtt{r}}$. Dans les PRUs, seuls changent les descripteurs de modules et leur distribution de probabilités de consommation de ressources. On y ajoute autant de dimensions que de nouvelles ressources. Le principal problème de notre approche est la grande taille de l'espace d'état, dûe aux dimensions supplémentaires introduites dans le modèle.

Mais, en exploitant certaines propriétés de la fonction de valeur des états, notamment sa croissance selon les ressources disponibles, le fait qu'elle soit bornée, et le facteur de ressource limitant (lorsqu'une ressource fait défaut, l'excès de l'autre ne fait pas augmenter la valeur) nous avons implémenté un algorithme permettant de calculer la fonction de valeur plus rapidement (les résultats expérimentaux sont dans le chapitre 10), en évitant d'évaluer et de stocker les états donc la valeur est située sur des paliers de valeur constante. De la même façon, nous représentons la fonction de valeur en évitant de garder ces mêmes paliers de valeur pour que le robot stocke dans son espace mémoire une fonction moins grande. Nous fournissons des résultats relatifs à ces expériences dans la dernière partie de ce mémoire.

Dans la partie suivante, nous allons exploiter la séparation du raisonnement en deux parties (présent/futur) pour calculer rapidement la fonction de gain espéré pour le reste de la mission de façon à pouvoir obtenir rapidement une politique à suivre localement en cas de changement ou d'imprévu dans la mission.

Troisième partie

Planification dynamique, via la décomposition

Chapitre 7

Planifier dans un environnement dynamique

Introduction

Dans un contexte réel, l'environnement auquel est confronté un robot n'est pas statique. Les choses changent, évoluent. Souvent, après avoir planifié un certain nombres d'actions dans la journée, un imprévu oblige l'agent à réviser son plan et à s'adapter aux nouvelles contraintes. Le système que nous avons présenté jusqu'ici, même s'il s'adapte parfaitement aux aléas inhérents à la consommation de ressources, ne permet pas de faire face à un changement dans la mission. La politique que nous calculons est optimale pour une mission donnée, elle devient obsolète dès le premier changement. Nous allons présenter dans ce chapitre les problèmes auxquels est confronté un agent évoluant dans un système dynamique et nous présenterons ensuite comment la dynamicité s'exprime dans l'environnement dans lequel le robot sera immergé. Nous allons ensuite proposer un modèle doublement adaptatif qui permettra au robot d'avoir un bon comportement si son environnement change, tout en consommant intelligemment les ressources restantes. Le chapitre suivant nous donnera l'occasion de mettre en œuvre des algorithmes permettant la génération d'un tel comportement adaptatif.

7.1 Un environnement dynamique

Des événements asynchrones peuvent apparaître dans un environnement dynamique. c'est-à-dire qu'à n'importe quel instant, le monde dans lequel le robot évolue peut être modifié. Ces modifications peuvent être plus ou moins fréquentes. Dans certains cas, quelques éléments seulement viennent perturber la disposition de l'environnement, on parle alors d'environnement faiblement dynamique. À l'extrême opposé, nous avons des systèmes dits chaotiques.

Ces changements peuvent avoir un impact sur la stratégie adoptée par l'agent lors de sa mission. Nous considérons qu'un événement asynchrone est détectable par le robot et qu'il va essayer d'en tenir compte dans sa stratégie. Nous voudrions avoir un **système adaptatif**, c'est-à-dire un robot capable d'adapter son comportement en fonction des changements dans l'environnement. Au mieux, nous pourrions espérer avoir une stratégie qui s'adapte à chaque changement de l'environnement et si possible que cette stratégie soit optimale au vue des critères considérés.

Évidemment, rien n'est si simple. Quand un événement survient, le robot doit s'adapter pour décider d'une action à entreprendre. Cette décision dépend de la stratégie adoptée par le robot. Or, le calcul même de la stratégie pose plusieurs problèmes :

- il n'est pas forcément possible de trouver une stratégie qui soit acceptable conjointement aux buts que l'on s'était fixés. Si par exemple le robot avait l'intention de faire des prélèvement sur un site dont l'accès est bloqué par un rocher, il doit abandonner ce site.
- Une stratégie peut exister, mais le temps nécessaire pour la calculer est trop long pour faire face au changement.

Il est donc nécessaire que le temps pour trouver la stratégie soit inférieur à la fréquence moyenne d'apparition des événements. Il existe pour de tels problèmes des architectures hybrides réactives et délibératives dont le principe est d'établir une solution optimale lorsque le temps le permet (la délibération) et de réagir au plus vite lorsque un imprévu ne laisse pas le temps pour un calcul de stratégie optimale. Il faut pouvoir trouver une solution, même non-optimale, avant une échéance donnée. Ces architectures répondent à un problème plus connu sous le nom de « temps réel ». Plusieurs architectures traitent déjà de planification temps réel sous incertitude : Guardian [Hayes-Roth, 1990], Phoenix [Howe *et al.*, 1990], CIRCA [Musliner *et al.*, 1993].

7.2 Description de notre environnement dynamique

Nous choisissons un scénario dans lequel le robot aura à effectuer une mission, sous forme de liste de PRUs. Pendant l'exécution de la mission une ou plusieurs tâches peuvent se rajouter ou se soustraire à cette liste de base. Notre objectif est de fournir au robot un système doublement adaptatif :

- l'**adaptation locale** permet de gérer l'incertitude liée à la consommation de ressources,
- l'**adaptation dynamique** permet de considérer les changements qui ont lieu dans la séquence de tâches futures.

Nous supposons que les nouvelles tâches qui vont apparaître pendant la mission sont des PRUs faisant partie d'un ensemble de tâches prédéfinies ; le robot possède donc un schéma d'exécution de ces tâches.

L'ordre de grandeur du temps au bout duquel la mission peut changer par ajout ou suppression de tâches est supposé supérieur à celui nécessaire pour effectuer un module donné. Le robot

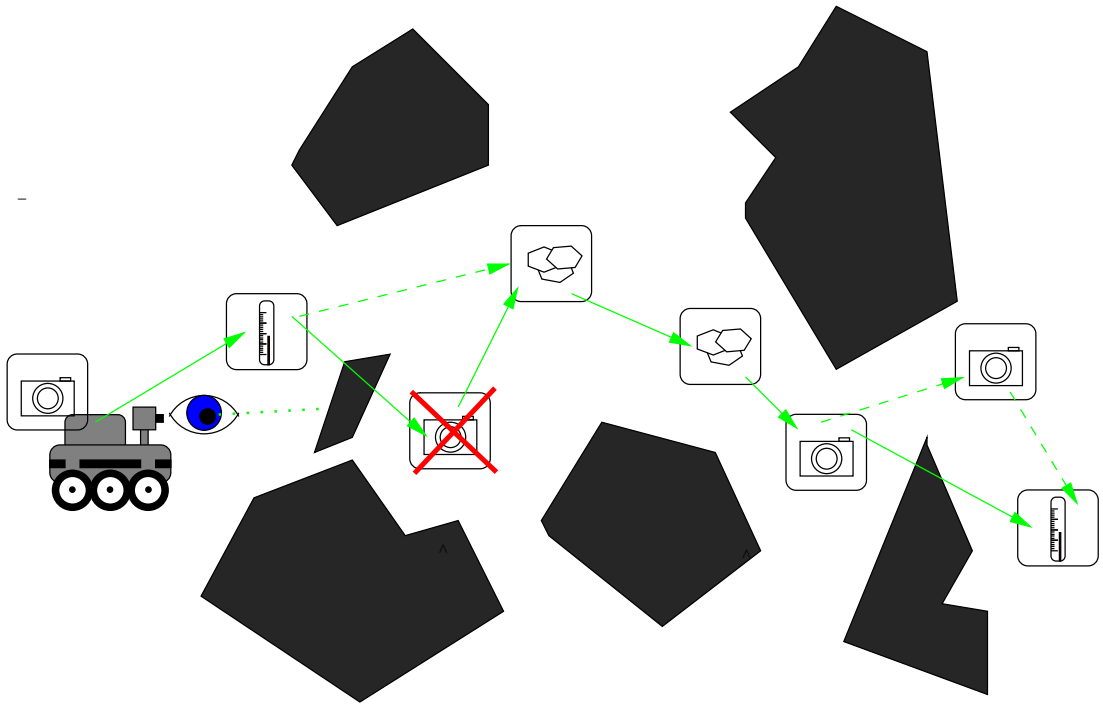


FIG. 7.1 – Des changements dans la mission

Sur la figure 7.1 nous représentons le chemin que pourrait suivre un robot pendant sa mission. Pendant qu'il prend une photo (tâche courante) il se rend compte qu'un rocher lui barre la route. Son plan initial tracé en lignes pleines, lui demandait de faire un relevé atmosphérique, puis de prendre une photo, puis de ramasser une roche. Comme le site à photographier n'est plus accessible, la mission change à ce moment. Le nouveau tracé de la mission est en pointillés. À la fin de la mission, une nouvelle tâche va apparaître, qu'il ne pouvait pas voir avant à cause d'un rocher.

possède ainsi à la fin de chaque module une vision stable de ce qu'il aura à faire dans le futur.

Nous allons rappeler comment le raisonnement progressif permet de s'adapter localement aux aléas de consommation de ressources avant de proposer un mécanisme permettant d'adapter son comportement lors d'un changement pendant la mission.

7.3 Les acquis : adaptation locale

Nous avons décrit dans les chapitres précédents comment un robot peut s'adapter à l'incertitude liée à la consommation de ressources lorsqu'on lui fournit une mission fixée à accomplir. Le comportement du robot est régi par une politique, calculée en maximisant le critère de gain espéré, lui même fonction des utilités définies pour chacune des PRUs qui constituent le reste de la mission. Ces utilités sont exprimées en fonction des ressources consommables restantes dans

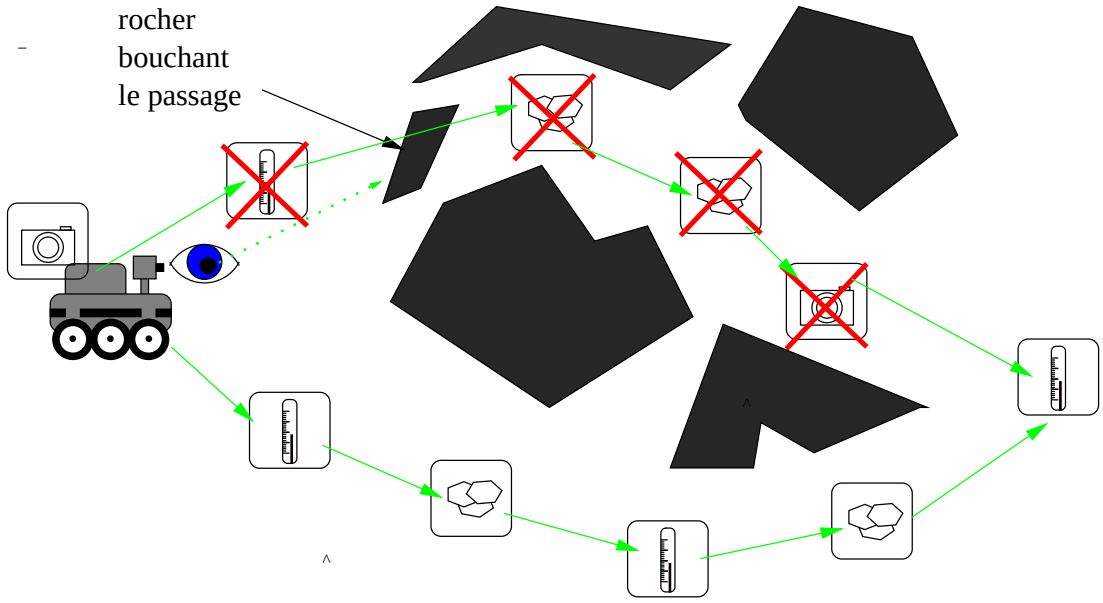


FIG. 7.2 – Changement total de mission

Sur la figure 7.2 nous représentons un changement total de mission. Le chemin de la mission de base n'est plus possible après la chute du rocher sur le chemin qui était prévu. Cette chute va demander au robot de revoir entièrement sa politique.

le robot et des qualités accumulées lors de l'exécution des modules au sein d'une même PRU.

La décision à prendre par le robot après exécution d'un module est dictée par la politique ainsi calculée, en fonction de l'état du robot. Donc, si la séquence de PRUs futures est connue, on calcule *a priori* une politique optimale pour cette mission et on confie ensuite au robot le soin de la suivre scrupuleusement, comme le montre la figure 7.3. Le fonctionnement classique des missions suit les étapes suivantes :

1. les concepteurs du robot font des mesures expérimentales pour évaluer le degré d'incertitude lié à la consommation de ressources pour tous les types de tâches que celui-ci est capable d'effectuer.
2. Les contrôleurs du robot prévoient une mission qu'ils modélisent comme une suite de PRUs.
3. La mission ainsi modélisée est transmise à un ordinateur qui calcule une politique qui tiendra compte des incertitudes liées à la consommation de ressources, ainsi que des tâches qui composent la mission.
4. La politique est transmise au robot qui effectuera la mission en choisissant simplement les actions proposées par la politique en fonction de son état.

C'est parce que la politique obtenue garantit de fournir à chaque instant une décision optimale vis à vis du critère de gain espéré compte tenu de l'incertitude liée à la consommation de

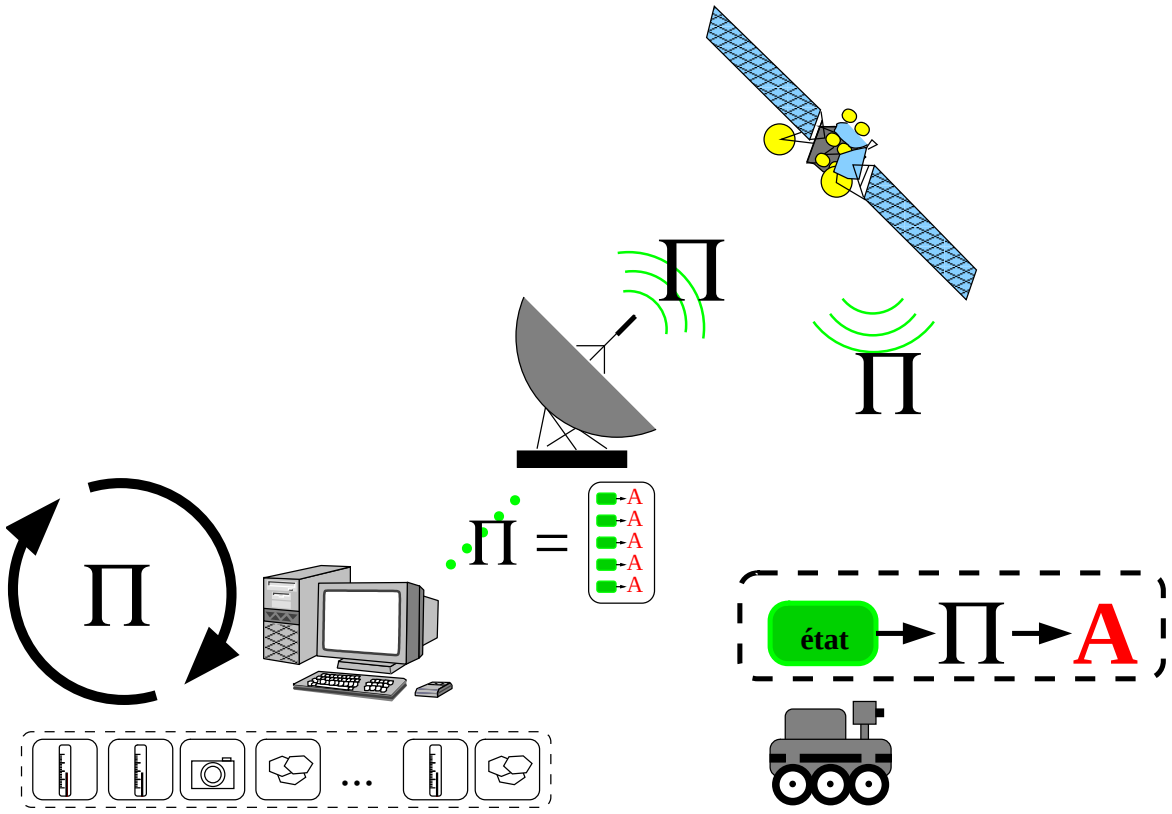


FIG. 7.3 – Envoi de la politique au robot et exécution de celle-ci

ressources que nous pouvons affirmer que le comportement du robot s'adapte à la consommation incertaine de ressources.

7.4 Adaptation dynamique

Cependant, l'adaptation à la consommation de ressources n'est pas suffisante pour tenir compte d'éventuelles modifications dans la séquence de tâches à effectuer dans le futur. Nous avons besoin d'une adaptation comportementale qui se situe à un autre niveau que celui de la politique suivie par le robot. La politique obtenue pour une séquence de tâches donnée devient obsolète dès que cette séquence est modifiée.

Notre problème a donc été non seulement de permettre au robot d'adapter son comportement lorsque la séquence de tâches est modifiée, mais aussi de trouver à quel niveau de raisonnement l'adaptation devait opérer. Il a fallu dans un premier temps repérer ce qui est modifié et ce qui reste valable dans le processus comportemental du robot et dans la mission lorsqu'il y a modification de mission. Politique, décision, action, gain espéré, fonction d'utilité, ressources

consommées, qualité, description de la mission, description des PRUs... Nous allons examiner ce qu'implique un changement dans la mission à un instant donné pour toutes ces notions.

Imaginons qu'à un instant $\mathtt{t}$ donné, pendant que le robot exécute un module $\mathsf{m}_{\mathtt{p},\mathtt{n},\mathtt{m}}$ de la $\mathsf{PRU}_{\mathtt{p}}$ de sa mission, une modification intervienne au niveau de la séquence de tâches composée des PRUs $\mathtt{p{+}1}$ à $\mathbf{P}$ (par exemple l'ajout de deux tâches et la suppression d'une tâche). Le robot possède à cet instant une politique à suivre, qui tient compte de la suite de l'ancienne mission et connaît son état, c'est-à-dire le montant de ressources qu'il lui reste, ainsi que le niveau qu'il vient d'exécuter. L'état courant reste le même : une modification sur la séquence de tâches futures n'affecte ni la tâche courante, ni les ressources restantes. Les types de PRUs que le robot est capable d'effectuer restent aussi les mêmes et leur description interne (niveaux, modules, distribution de probabilités de consommation de ressources) est inchangée. Par contre, le gain espéré pour le reste de la mission change. La politique est calculée à partir de ce gain espéré, donc les actions liées aux états ne sont plus forcément optimales, la politique n'est plus appropriée à la situation.

Nous pourrions recalculer entièrement la politique pour la nouvelle mission, ce qui revient à une réadaptation totale du contrôle de la mission. Cette méthode radicale doit être évitée puisque le temps nécessaire au calcul d'une politique optimale peut être long et même dépasser le temps nécessaire pour effectuer un module. De plus, il n'est pas nécessaire de connaître l'action liée à un état concernant une PRU dans le futur, puisque l'action à prendre ne concerne que la PRU courante et que la mission est susceptible de changer une nouvelle fois. Plutôt que d'avoir un contrôle total sur la mission, nous allons opter pour un contrôle local de celle-ci, en ne recalculant la politique que pour la PRU dans laquelle il est en train d'évoluer, en tenant compte de l'état courant et du reste de la mission. Nous justifions cette approche par le fait que le robot n'a pas besoin de savoir ce qu'il fera plus tard pour prendre une décision maintenant, mais seulement de connaître l'impact d'une décision dans le présent sur le futur.

La politique recalculée localement doit être disponible suffisamment rapidement pour que si un changement intervient dans la mission, il puisse prendre une décision à la fin de ce module. Le temps nécessaire pour recalculer la politique locale doit donc être inférieur d'une part à la fréquence d'apparition ou de disparition de nouvelles tâches dans la mission et d'autre part inférieur au temps nécessaire pour effectuer un module. Ce besoin de rapidité pour l'adaptation au changement dans la mission nous oblige à ne pas forcément calculer une politique locale optimale. Nous devons trouver un compromis entre la valeur de la politique obtenue par rapport à l'optimale et le temps nécessaire pour la calculer.

La technique du coût occasionné nous permet de calculer une politique pour une seule PRU en tenant compte à la fois de l'impact local lié à la consommation de ressources et aux conséquences futures liées à la consommation locale de ressources. Cette technique permet d'agir localement tout en mesurant les conséquences des actions prises sur le futur. Nous allons donc utiliser ce

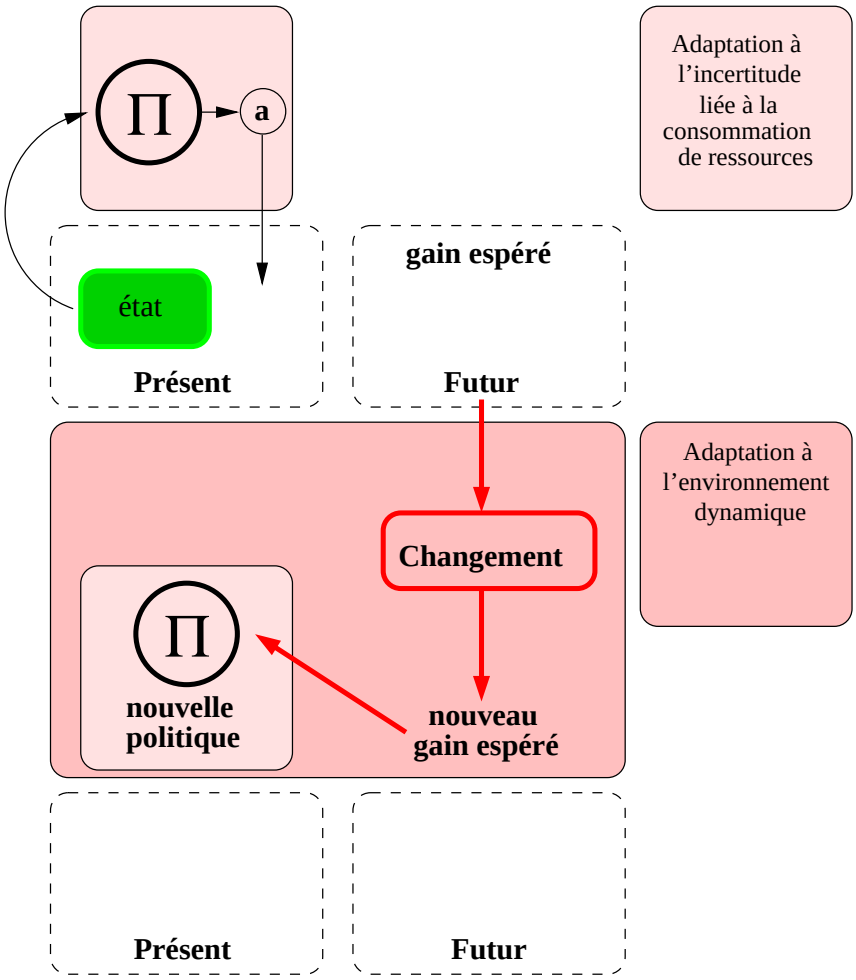


FIG. 7.4 – Un système doublement adaptatif

Un changement dans la mission implique un changement dans le futur. Ceci a des répercutions sur la politique, directement liée au gain espéré dans le futur. Une nouvelle politique est calculée avec la nouvelle fonction de gain espéré pour s'adapter au changement d'environnement. Elle tiendra compte elle aussi des incertitudes liées à la consommation de ressources.

concept pour adapter le comportement du robot aux changement futur. En **mesurant rapidement le coût occasionné** par une consommation de ressource, nous pourrons décider si oui ou non l'exécution du module suivant est intéressant vis à vis du critère de gain espéré. La figure 7.4 schématise le fonctionnement du **système doublement adaptatif**.

Conclusion

Dans ce chapitre nous avons exposé le problème d'un robot qui doit effectuer une mission où la consommation de ressource est incertaine d'une part et où la séquence de tâches peut

évoluer pendant que le robot effectue sa mission. Après avoir présenté les problèmes existants pour la planification en milieu dynamique, nous avons décrit l'environnement dynamique dans lequel évolue le robot. Finalement, nous exposons notre objectif : obtenir un système doublement adaptatif permettant au robot de gérer sa consommation de ressources malgré les incertitudes et réagir face aux changements occurrents dans la séquence de tâches futures. Dans le chapitre suivant, nous allons mettre en œuvre des techniques permettant de s'adapter à ces changements dynamiques.

Chapitre 8

Système d'adaptation à l'évolution dynamique de l'environnement

Introduction

Nous avons exposé dans le chapitre précédent le problème auquel nous sommes confrontés, à savoir créer un système doublement adaptatif permettant de gérer les ressources à consommer malgré l'incertitude pendant une mission lors de laquelle pourraient intervenir certaines modifications. L'idée générale de notre approche est de générer rapidement une fonction de gain espéré approchée qui va permettre à l'agent de calculer localement une politique pour la PRU courante. Nous proposons ici une approche qui repose sur la recomposition d'une fonction convexe à partir de plusieurs morceaux de courbes provenant des profils de performance des tâches qui composent la mission.

Nous montrerons dans la partie suivante, chapitre 11, que cette recomposition est rapide et qu'elle permet d'obtenir une politique non optimale. mais grâce à celle-ci l'agent adoptera un comportement acceptable. Certains résultats de ce chapitre ont donné lieu à une publication internationale [Le Gloannec *et al.*, 2005a].

8.1 Objectif

Notre objectif est de calculer le plus rapidement une approximation de la fonction de valeur espérée pour le reste de la mission. Nous avons vu comment la calculer avec une seule[44] ressource consommable dans le chapitre 5. La complexité de cet algorithme est $\mathbf{C}.\mathbf{r}_{\mathbf{P}}^{max}.\frac{\mathbf{P}.(\mathbf{P}-1)}{2}$.

[44] Dans ce chapitre, nous considérons de nouveau qu'il n'y a qu'une seule ressource consommable.

8.1.1 Rappels sur le calcul de la fonction de valeur optimale V_1^*.

La connaissance de la fonction de valeur $V_1^* : \mathtt{R} \rightarrow \mathbb{R}$ pour le reste de la mission (les PRUs 1 à $\mathbf{P}$) est suffisante pour calculer la fonction de valeur pour la PRU_0 courante. Sur la figure 8.1, nous schématisons le principe de fonctionnement de l'algorithme 7 page 85.

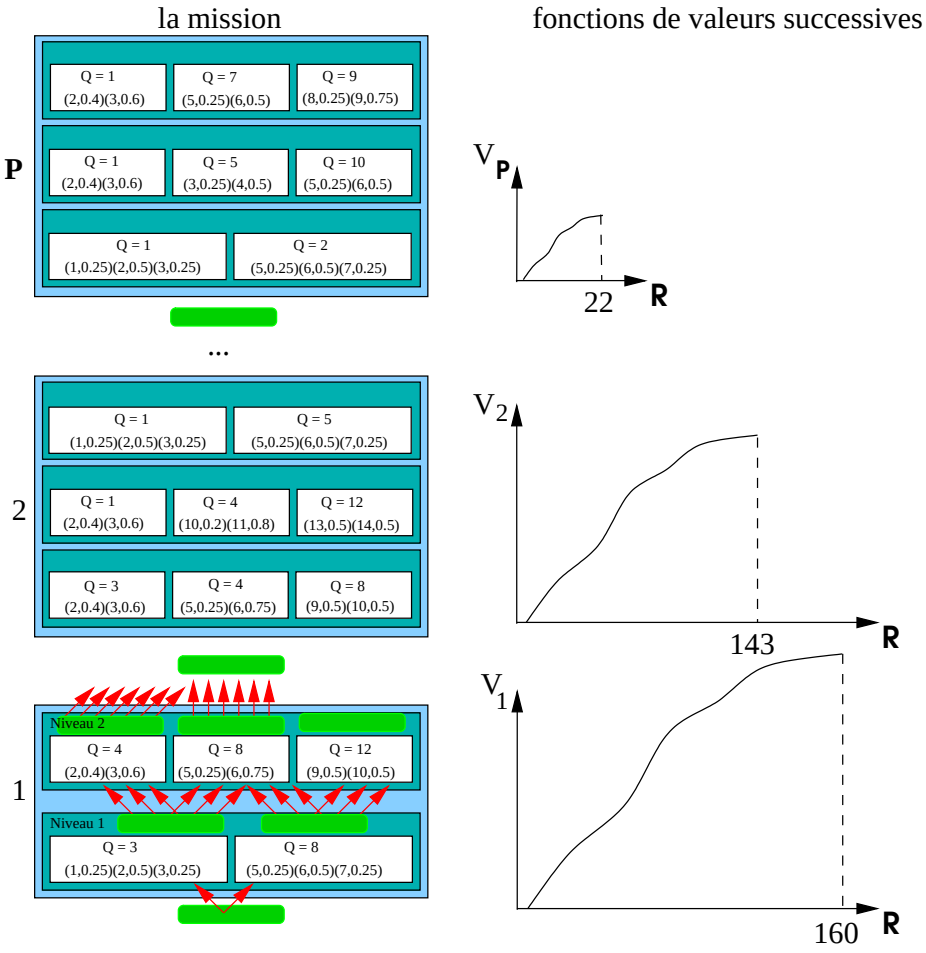


FIG. 8.1 – Calcul schématisé de V_1^* pour une mission avec une seule ressource.

Explications *pour la figure 8.1 : on présente sur cette figure l'algorithme de calcul de la fonction de valeur espérée pour les PRUs* 1 *à* $\mathbf{P}$. *Le schéma se lit de haut en bas. A gauche, les PRUs de la mission sont classées de la dernière à la première. A droite, on représente les fonctions de valeurs : la première fonction concerne seulement la* $\mathsf{PRU}_\mathbf{P}$. *On commence par calculer la fonction de valeur* $V_\mathbf{P}$ *pour la* $\mathsf{PRU}_\mathbf{P}$. 22 *unités de ressources suffisent à obtenir à coup sûr un résultat maximal, donc la fonction de valeur* $V_\mathbf{P}$ *est exprimée sur l'intervalle* [0..22]. *Ensuite, on calcule* $V_{\mathbf{P}-1}$ *à partir de* $V_\mathbf{P}$. $V_{\mathbf{P}-1}$ *est définie sur un support environ deux fois plus grand que celui de* $V_\mathbf{P}$. *En suivant l'algorithme 7, on calcule* V_1 *avec la* PRU_1 *à partir de* V_2. *Le support de* V_1 *est agrandi de* 17 *unités de ressources par rapport à* V_2 *puisqu'en consommant* 17

ressources, l'agent est sûr d'obtenir un gain maximal dans cette PRU_1 *(pour s'en convaincre, il suffit d'additionner les consommations des trois modules de droite de la* PRU_1 *:* $10 + 7 = 17$*). Ensuite, si la mission ne change pas, on obtient de la même façon la fonction de valeur pour la* PRU_0 *courante.*

Cependant, le temps nécessaire à l'obtention de cette fonction de valeur optimale peut être trop long dans un environnement dynamique. Nous avons donc eu recours à des techniques d'approximation de cette fonction V_1.

8.1.2 Principe général de l'approximation

Nous proposons dans ce chapitre trois méthodes pour l'approximation de cette fonction V_1. Ces méthodes sont basées sur un même principe : l'approximation par ajouts successifs de fonctions élémentaires. Tout ceci s'inspire des méthodes d'approximateurs de fonctions et des méthodes de décompositions vues dans le chapitre 3. Chacune de ces fonctions élémentaires représente le gain espéré local à chaque PRU.

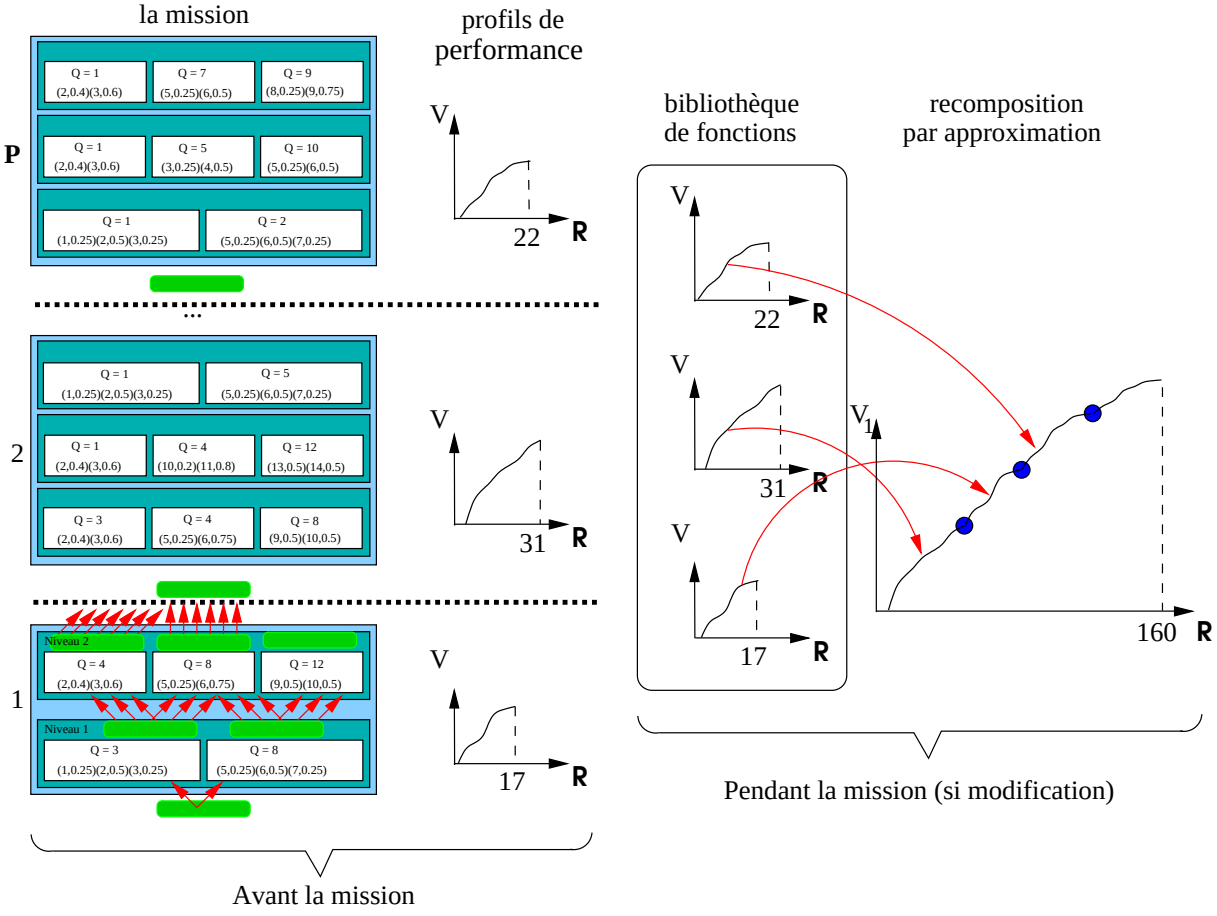


FIG. 8.2 – Le principe général de la méthode d'approximation de la fonction de valeur V_1.

Explications *pour la figure 8.2 : avant la mission, on calcule pour chaque type de PRU possible un profil de performance. Chaque profil de performance est le gain espéré pour une PRU donnée en fonction des ressources qui lui seront éventuellement attribuées, indépendamment du contexte de la mission. Leur support est donc restreint. Ils sont stockés dans une bibliothèque de fonctions avant la mission. Si pendant la mission il y a une modification dans la séquence de PRUs, la fontion de gain espéré globale V_1 sera recalculée à partir de ces profils de performance mises bout à bout.*

La première méthode dite "naïve", consiste à mettre bout à bout toutes ces fonctions élémentaires. La deuxième méthode en est une amélioration. La troisième et dernière méthode présentée en fin de chapitre apporte une amélioration supplémentaire, mais le calcul de la fonction de valeur est plus lent.

Naturellement, la politique obtenue pour la PRU_0 sera différente si on utilise V_1^* ou une approximation de V_1. Nous proposons de valider notre approche par une comparaison des Q-valeurs associées à la politique obtenue grâce à l'une ou l'autre des méthodes

Notation : puisque la fonction que nous calculons sera toujours $V_1 : \mathtt{R} \rightarrow \mathbb{R}$, nous la notons $V : \mathtt{R} \rightarrow \mathbb{R}$. $V^* : \mathtt{R} \rightarrow \mathbb{R}$ désigne la fonction de valeur optimale tandis que $V^{\sim}_{methode} : \mathtt{R} \rightarrow \mathbb{R}$ est une fonction approchée par la méthode `methode`.

8.1.3 Une idée d'approximation

Après avoir effectué plusieurs calculs de fonction de valeur pour des missions composées de PRUs, nous avons constaté que cette fonction $V^* : \mathtt{R} \rightarrow \mathbb{R}$ à une allure « globalement » convexe (voir figure 8.3).

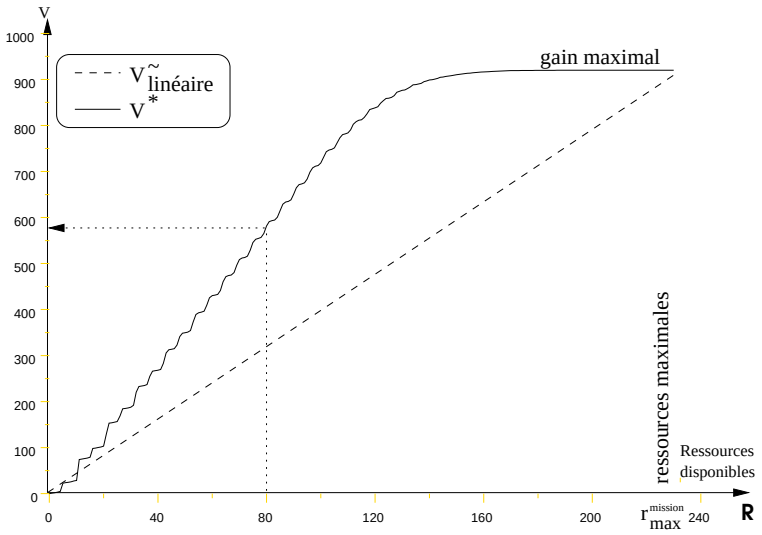


FIG. 8.3 – V^* et $V^{\sim}_{lineaire}$ pour une mission donnée.

Ceci s'explique par le fait que le contrôle de consommation de ressources via le raisonnement progressif ressemble au contrôle de la qualité retournée par algorithme anytime, comme nous l'avons vu dans le chapitre 4. Les premières améliorations sont plus fortes que les suivantes. La courbe en trait continu est le résultat d'un calcul de la fonction V^* pour une mission composée d'une vingtaine de PRUs. La valeur espérée $V^* : \mathtt{R} \rightarrow \mathbb{R}$ se lit de cette façon : si l'agent dispose de 80 unités de ressources pour le reste de la mission, il peut espérer gagner 575.

L'approximation la plus naïve est de tracer une droite en partant de l'origine (si l'agent dispose de 0 ressources il n'obtiendra aucun résultat) pour aller jusqu'au point $(\mathtt{r}_{\max}^{mission}, V_{\max})$. Cette droite figure en pointillés sur la figure 8.3. Rappelons qu'il est possible de connaître très rapidement ce point de gain maximal : les ressources correspondent à la somme de l'ensemble des ressources que l'on peut consommer au maximum dans toutes les tâches. Le gain espéré maximal correspondant est la somme de toutes les qualités maximales que l'on peut obtenir après chaque tâche.

$$\mathtt{r}_{\max}^{mission} = \sum_{\mathtt{p}=1}^{\mathbf{P}} \sum_{\mathtt{n}=1}^{\mathtt{N_p}} \max_{\mathtt{m} \in \mathtt{M_n}} \left(\max_{\mathtt{r}} (\mathtt{r} \in Dom(\mathcal{P}r_{\mathtt{p,n,m}})) \right) \tag{8.1}$$

Ceci permet d'approcher la fonction de gain espéré très grossièrement. Nous appelons[45] cette approximation **l'approximation linéaire** et nous notons cette fonction $V_{lineaire}^{\sim}$.

$$\begin{array}{rccl} V_{lineaire}^{\sim} : \mathtt{R} & \rightarrow & \mathbb{R} & \\ \mathtt{r} & \rightarrow & \mathtt{r}.\dfrac{V_{\max}}{\mathtt{r}_{\max}^{mission}} & \end{array}$$

Nous allons affiner cette méthode en recherchant une courbe convexe dont l'allure se rapprochera plus fidèlement de la fonction V^*. Mais d'abord, nous présentons dans la section suivante comment valider une méthode d'approximation de fonction de gain espéré pour le raisonnement progressif, après quoi nous exposerons dans ce chapitre différentes méthodes pour obtenir une telle approximation.

8.2 Validation de l'approche

Notre approche consiste donc à calculer une politique pour la PRU_0 courante en se basant sur une fonction de gain espéré approchée $V^{\sim}$. Nous allons montrer que la valeur des actions dictées par cette politique est proche de celles de la politique optimale.

Il est possible dans notre problème de calculer la fonction de gain espéré V^* qui permet d'obtenir une politique optimale par programmation dynamique. Ce calcul peut prendre plusieurs

[45] Cette approximation ne fait pas partie de nos trois méthodes.

minutes pour certaines missions. Néanmoins, cela va nous faciliter la tâche pour la comparaison des politiques obtenues. Puisque nous disposons de la politique optimale et d'une politique sous-optimale, nous comparerons pour tous les états possibles de la PRU_0 la valeur de l'action qui sera prise par le robot qui suit chacune de ces deux politiques. Cette comparaison se base sur les Q-Valeurs de la politique *optimale* associées à chaque paire état, action. Elle permet de quantifier l'erreur commise par le robot, alors que la simple comparaison de la nature des actions prises donne une mesure qualitative mathématiquement inexploitable.

Une Q-valeur est la valeur associée à une action dans un état donné, étant donné une politique π suivie :

$$\mathcal{Q}^{\pi}(\mathbf{s},\mathbf{a}) \quad = \quad \sum_{\mathbf{s}'\in\mathcal{S}} \mathcal{P}r(\mathbf{s}'|\mathbf{a},\mathbf{s}) \times V^{\pi}(\mathbf{s}') \tag{8.2}$$

Nous expliquons sur les schémas qui suivent notre démarche expérimentale qui se décompose en quatre étapes :

1. calcul des gains espérés V^* et $V^{\sim}_{methode}$ pour une même mission,
2. calcul des politiques π^* et $\pi^{\sim}_{methode}$ pour la même PRU_0 locale, en utilisant le gain espéré correspondant,
3. pendant le calcul de π^*, garder **toutes** les Q-valeurs associées à chaque couple $(\mathbf{s},\mathbf{a})$,
4. pour chaque état possible $\mathbf{s}$ de la PRU_0 locale, mesure de la différence entre la Q-valeur optimale et la Q-valeur dans la politique optimale associée à l'action liée à cet état dans la politique $\pi^{\sim}_{methode}$.

Le calcul des Q-valeurs se fait uniquement sur la politique optimale. Ce sont ces seules Q-valeurs qui nous permettront de mesurer la fiabilité des politiques approchées obtenues. La politique optimale pour la PRU_0 se calcule en évaluant les états de cette PRU via l'équation de Bellman 5.9. La valeur des actions $\mathbf{M}$ est celle du gain espéré optimal pour le reste de la mission.

Le calcul de chaque politique approchée se fait avec le même algorithme quelle que soit la méthode utilisée pour calculer le gain espéré : nous utilisons la programmation dynamique et évaluons les états de la PRU courante. Le seul calcul qui diffère est la valeur associée à une action $\mathbf{M}$ de changement de PRU : elle est égale au gain espéré calculé par la méthode approchée. Nous obtenons ainsi les politiques π^* et $\pi^{\sim}_{methode}$ pour chaque méthode de calcul de gain espéré.

Note : nous ne calculons pas les Q-valeurs pendant le calcul des politiques $\pi^{\sim}_{methode}$.

Finalement pour une méthode donnée, nous comparons sur l'ensemble des états possibles de la PRU_0 courante les actions dictées par la politique optimale et les actions dictées par la politique approchée $\pi^{\sim}_{methode}$. Nous recherchons la Q-valeur dictée par l'action de la politique optimale (calculée dans l'étape 2). Nous comparons alors cette Q-valeur à celle obtenue si l'agent effectue l'action dictée par la politique $\pi^{\sim}_{methode}$. La mesure d'erreur se fait par l'équation 8.3 :

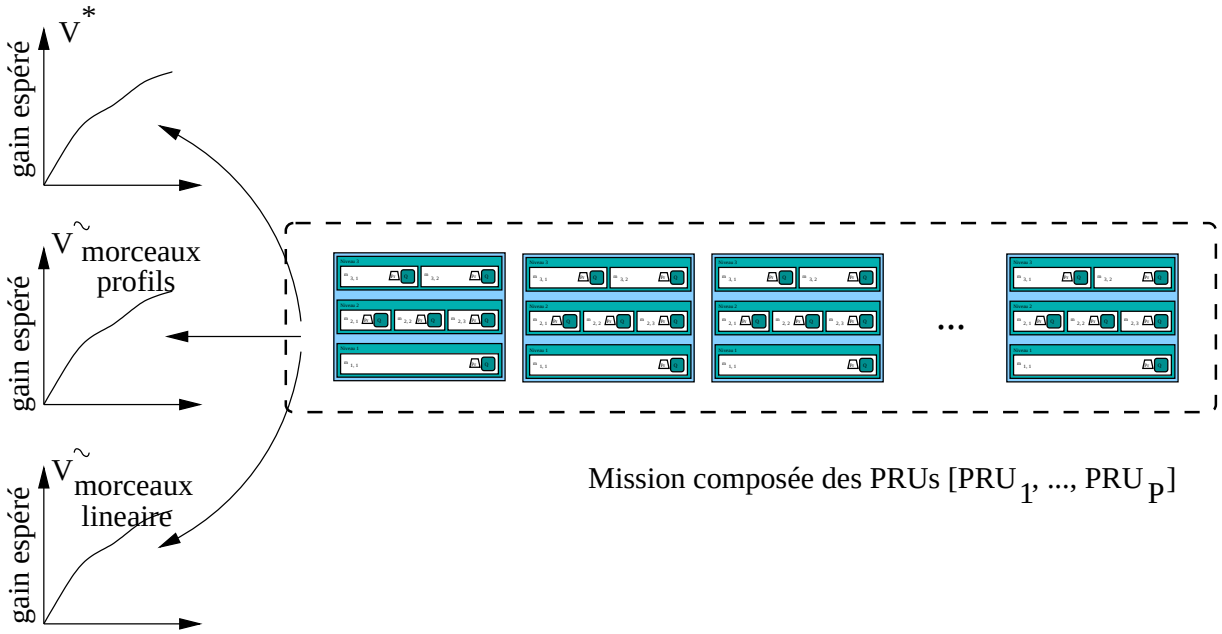


FIG. 8.4 – Première étape : Calcul des gains espérés avec les différentes méthodes.

Explications *pour la figure 8.4 : pour une mission donnée, nous calculons via différentes méthodes les fonctions de gain espérés, à savoir la fonction de gain espéré optimale* V^**, les approximations* $V^{\sim}_{morceauxlineaire}$, $V^{\sim}_{morceauxprofils}$, *et l'approximation linéaire simple* $V^{\sim}_{lineaire}$. *Nous présentons ces fonctions dans la section suivante.*

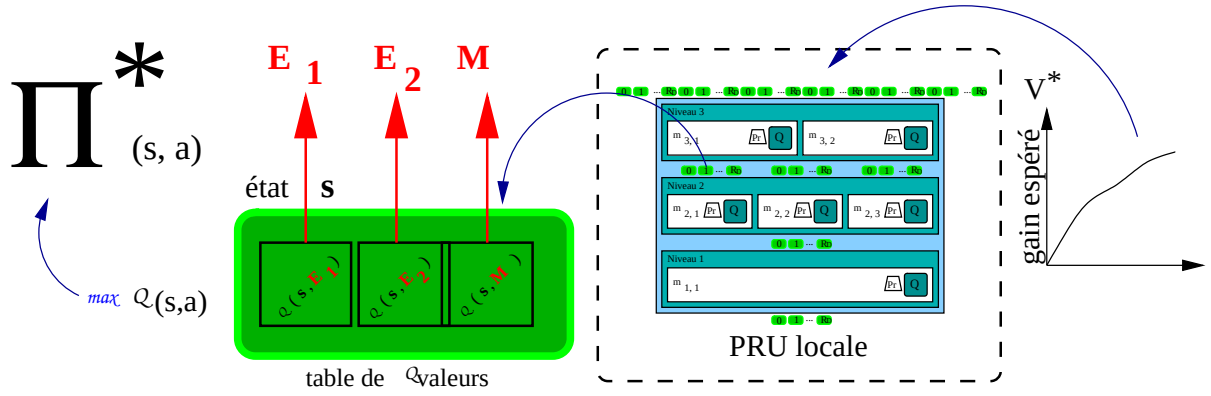


FIG. 8.5 – Deuxième étape : Calcul des Q-valeurs dans la politique optimale

Explications *pour la figure 8.5 : à partir de la fonction de gain espéré optimale calculée précédemment, nous calculons la politique optimale pour la PRU courante, puis nous conservons toutes les Q-valeurs associées aux différentes paires état action* $(\mathbf{s}, \mathbf{a})$. *Ces Q-valeurs serviront de mesure étalon pour évaluer la qualité des politiques obtenues par approximation.*

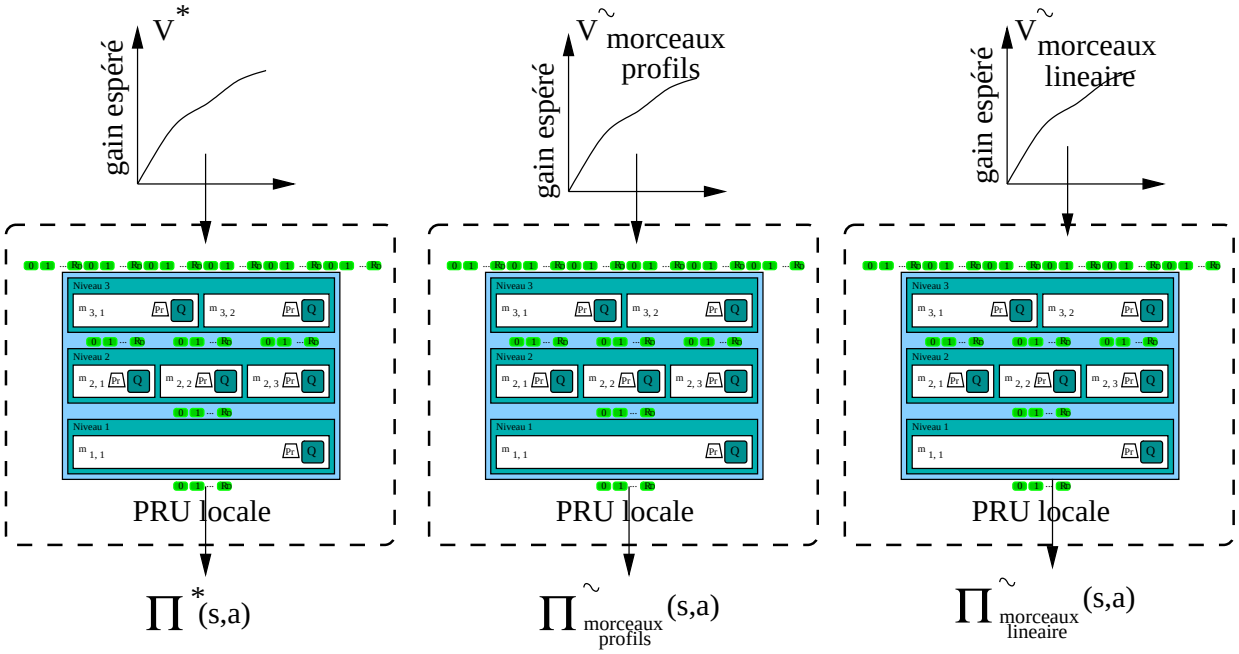


FIG. 8.6 – Troisième étape : Calcul des politiques sous-optimales.

Explications *pour la figure 8.6 : nous calculons pour la même PRU locale une politique en se basant sur les différentes fonctions de gain espéré. Nous obtenons chaque fois une politique approchée. Nous ne conservons pas les valeurs propagées par ces politiques approchées qui n'ont pas de réelles signification mathématique. De ce calcul, nous ne conservons que les politiques sous-optimales obtenues, c'est-à-dire les paires état action* $(\mathbf{s}, \mathbf{a})$.

$$\forall \mathbf{s}, Erreur_{\mathcal{Q}}(\mathbf{s}) \quad = \quad \mathcal{Q}(\mathbf{s}, \pi^*(\mathbf{s})) - \mathcal{Q}(\mathbf{s}, \pi^{\sim}_{methode}(\mathbf{s})) \tag{8.3}$$

De cette façon, si les actions dictées par les deux politiques sont les mêmes, l'erreur est nulle. Si les deux actions sont différentes, le système de comparaison par Q-valeur permet de mesurer la pertinence du choix effectué. En effet, deux actions différentes peuvent conduire à des résultats proches au niveau du gain obtenu. Comparer seulement les actions dictées par chaque politique ne suffit pas : certaines actions différentes peuvent au final[46] conduire à un résultat dont le gain est soit proche, soit éloigné de l'optimal. Le système de comparaison par Q-valeurs permet de mesurer cet éloignement. Une action différente de l'action dictée par la politique optimale est une bonne action si le gain qui en résulte est proche de ce qui aurait dû être obtenu avec l'action optimale. Nous allons utiliser cette mesure par comparaison de Q-valeurs dans le chapitre 11.

Dans la section suivante, nous présentons la méthode que nous avons mise au point pour approcher rapidement la fonction de gain espéré sans passer par la programmation dynamique.

[46] Après exécution de l'ensemble du plan.

8.3 Première méthode de décomposition/recomposition de V

Cette première méthode d'approximation de V est l'application directe de ce qui a été présenté en début de chapitre. Avant la mission, on va calculer pour chaque PRU une fonction de valeur *locale* que l'on appelle profil de performance, puis, dès qu'un changement intervient, on se sert de ces profils comme fonctions de base pour approcher V^*.

Note : le robot martien est capable d'effectuer plusieurs types de tâches différentes comme prendre des photos sur un site, de ramasser du minerai ou faire des relevés atmosphériques par exemple (voir figure 8.7). Il est inutile de calculer plusieurs fois le même profil de performance pour le même type tâche. Nous notons $\alpha, \beta, \gamma, \ldots$ les différentes tâches que le robot sait faire. Pour préciser le type de tâche qu'aura à effectuer le robot dans la mission, nous mettons le type de tâche en exposant : PRU_8^{β}. Ceci indique que la huitième tâche de la mission consiste à ramasser un minerai.

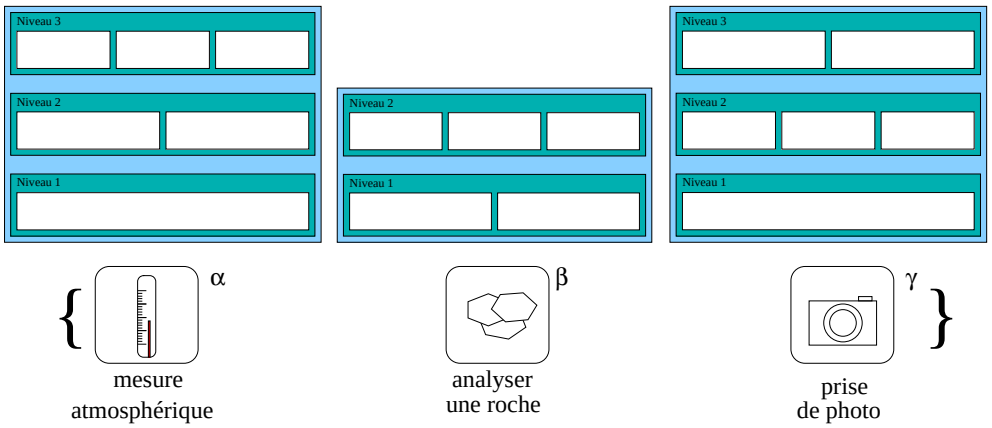


FIG. 8.7 – Exemples de tâches faisables.

Notre méthode va être séparée en deux parties :

- la **décomposition** se fait avant la mission, elle consiste à extraire une fonction de gain espéré f^{α} pour chaque type de PRU^{α} que le robot est censé savoir accomplir ; nous appelons ces f^{α} des profils de performance
- la **recomposition** a lieu pendant la mission si celle-ci subit des changements ; on se sert alors des fonctions de gain espéré locales f^{α} pour recomposer une fonction de gain espéré globale $V^{\sim}$ qui nous permettra de recalculer une politique locale pour la PRU_0 dans laquelle le robot se trouve lorsque le changement dans la mission a lieu.

8.3.1 Décomposition : calcul de profils de performances pour chaque PRU^α

Avant la mission, nous calculons pour chaque type de PRU^α un profil de performance f^α qui n'est autre que la valeur espérée à l'entrée d'une mission composée d'une unique PRU^α. Cette fonction f^α va avoir comme support l'espace des ressources consommables par cette PRU de type α, $[0,..,\mathtt{r}_{\mathtt{p}^\alpha}^{max}]$, $\mathtt{r}_{\mathtt{p}^\alpha}^{max}$ étant le maximum de ressources que l'on peut dépenser dans cette PRU^α. Nous notons $\mathcal{F}$ l'ensemble des profils de performance. Cette fonction f^α est croissante et bornée par :

$$f^\alpha(\mathtt{r}_{\mathtt{p}^\alpha}^{max}) \quad = \quad \sum_{\mathtt{n}=1}^{\mathtt{N}_{\mathtt{p}^\alpha}} \max_{\mathtt{m}\in\mathtt{M}_\mathtt{n}} \mathbf{Q}_{\mathtt{p},\mathtt{n},\mathtt{m}}$$

Nous savons déja calculer un profil de performance : il suffit de calculer pour toutes les valeurs de ressources dans $[0,..,\mathtt{r}_{\mathtt{p}^\alpha}^{max}]$ la valeur espérée obtenue en exécutant cette unique PRU^α, avec un algorithme de programmation dynamique classique. L'algorithme 5 page 83 nous permet par exemple de calculer un profil de performance.

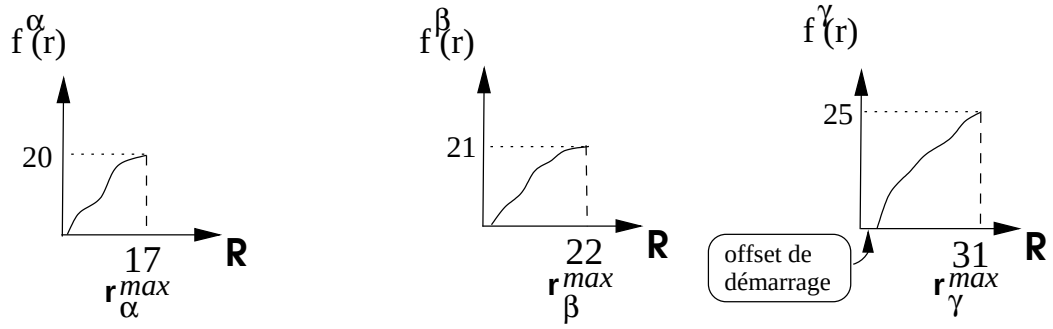


FIG. 8.8 – Etape 1 : calcul de profils de performance pour 3 types de PRUs.

Explications *pour la figure 8.8 : nous avons calculé pour chaque type de* PRU^α *un profil de performance* $f^\alpha : \mathtt{R} \rightarrow \mathbb{R}$. *Chaque profil de performance indique la valeur que l'on peut espérer gagner en allouant une certaine quantité de ressources à ce type de PRU. Le calcul du profil de performance se fait indépendamment de la mission, pour une seule PRU. Ces fonctions sont croissantes.*

Une fois calculée puis enregistrées, ces fonctions sont triées par ordre de meilleur rapport $\frac{f^\alpha(\mathtt{r}_{\mathtt{p}^\alpha}^{max})}{\mathtt{r}_{\mathtt{p}^\alpha}^{max}}$. Dans notre exemple, PRU^α seront mises avant les PRU^β. Attention, ceci ne veut absolument pas dire que l'ordre des PRUs dans la mission va changer. Ce tri resservira pour calculer rapidement l'approximation de $V*$.

Note : nous allons calculer la taille de l'ensemble des états qu'il faudra évaluer pour calculer tous les profils de performance. Le calcul du profil de performance de chaque tâche se fait indépendamment des autres tâches de la mission. Ainsi pour un type de PRU^α donné, nous avons un espace d'états local de taille :

$$|\mathcal{S}_{\mathtt{p}^\alpha}| \quad \leq \quad \mathtt{r}_{\mathtt{p}^\alpha}^{max}.(1+\sum_{\mathtt{n}=1}^{\mathtt{N}_{\mathtt{p}^\alpha}} \prod_{\mathtt{n}'=1}^{\mathtt{n}} |\mathsf{m}_{,\mathtt{p}^\alpha,\mathtt{n}'}|)$$

Sur une mission composée de **P** PRUs nous avons un nombre d'états inférieur ou égal à :

$$|\mathcal{S}| \leq \sum_{\mathtt{p}^\alpha \in \mathsf{mission}} \left(\mathtt{r}^{max}_{\mathtt{p}^\alpha} . \left(1 + \sum_{\mathtt{n}=1}^{\mathtt{N}_{\mathtt{p}^\alpha}} \prod_{\mathtt{n}'=1}^{\mathtt{n}} |\mathtt{m}_{,\mathtt{p}^\alpha,\mathtt{n}'}|\right)\right) \tag{8.4}$$

(fin de note)

Une fois cette décomposition effectuée avant la mission, le robot est prêt à subir des changements pendant la mission.

8.3.2 Recomposition : approximation de V^* par $V^{\sim}_{somme\,profils}$.

Pour recomposer une approximation de la fonction de valeur, nous disposons de plusieurs profils de performance f^α et du nombre $\mathtt{p}_\alpha$ de PRUs de type α dans la mission. En triant la liste des profils de performance par ordre de rapport, nous savons que les améliorations les plus fortes se situeront en début de fonction recomposée, et les plus faibles à la fin. La fonction de gain espéré $V^{\sim}_{somme\,profils}$ est une somme de tous les profils de performance triés. Nous les mettons bout à bout, comme sur la figure 8.9. Pour cette première méthode, nous nous basons sur l'intuition suivante : si l'agent dispose exactement de la quantité de ressources suffisante pour réaliser les tâches qui rapportent globalement le plus, alors il aura intérêt à n'effectuer que ces tâches. C'est pour cette raison que les profils de type α, qui ont le meilleur rapport dans l'exemple ci-dessous, sont tous placés en premier.

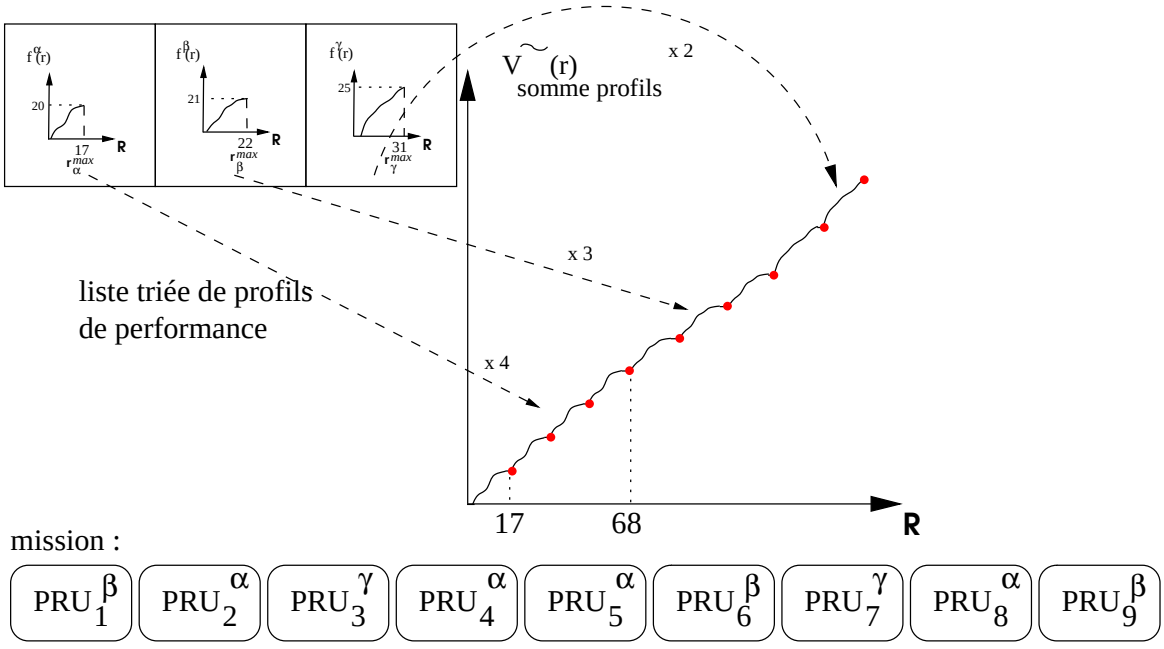


FIG. 8.9 – Etape 2 : reconstitution de $V^{\sim}_{somme\,profils}$ pour approcher V^*.

Explications *pour la figure 8.9 : la mission est composée de 4 PRUs de type α, 3 PRU$^\beta$ et 2 PRU$^\gamma$. Pour recomposer la fonction $V^{\sim}_{somme\,profils}$, on commence au point $(0,0)$, et on ajoute une fois le profil f^α. On se place ensuite sur le point $(f^\alpha(\mathtt{r}_{\mathtt{p}^\alpha}^{max}), \mathtt{r}_{\mathtt{p}^\alpha}^{max}) = (17,20)$ puis on ajoute une deuxième fois le même profil. Cela doit être fait 4 fois puisqu'il y a 4 PRUs de type α. A partir du point $(4*17, 4*20)$ on ajoute de la même façon 3 fois f^β puis finalement 2 fois f^γ.*

Le support de la fonction reconstituée $V^{\sim}_{somme\,profils}$ est strictement le même que celui de la fonction optimale : c'est l'intervalle $[0..\sum_{\mathtt{p}=1}^{\mathbf{P}} \mathtt{r}_{\mathtt{p}}^{max}]$. Autre remarque : les deux courbes sont égales à leurs extrémités. L'algorithme 13 permet de recomposer $V^{\sim}_{somme\,profils}$.

Données : mission = $[\mathsf{PRU}_1, \ldots \mathsf{PRU}_{\mathbf{P}}]$, $\mathcal{F}$
$\mathtt{r}_{allouee} = 0$ //ressource déjà allouée pour la mission.
$V_{prec} = 0$ //dernière valeur sur la courbe.
tant que $\mathtt{r}_{allouee} <= \mathtt{r}_{mission}^{max}$ **faire**
  $\alpha = \underset{\alpha}{\operatorname{argmax}} \left(f^\alpha(\mathtt{r}_{\mathtt{p}^\alpha}^{max}), f^\alpha \in \mathcal{F}\right)$ //type de la meilleure PRU
  $n_\alpha = \mathbf{nb}(\mathsf{PRU}^\alpha \in mission)$ //nombre de PRU$^\alpha$ dans la mission
  pour $i \in [1, \ldots, n_\alpha]$ **faire**
    pour $\mathtt{r} \in [\mathtt{r}_{allouee}, \ldots, \mathtt{r}_{allouee} + \mathtt{r}_{\mathtt{p}^\alpha}^{max}]$ **faire**
      $V^{\sim}_{sommeprofils}(\mathtt{r}) = V_{prec} + f^\alpha(\mathtt{r})$ //Copie des valeurs du meilleur profil.
    finpour
    $\mathtt{r}_{allouee} += \mathtt{r}_{\mathtt{p}^\alpha}^{max}$
    $V_{prec} = V^{\sim}_{somme\,profils}(\mathtt{r}_{allouee})$ //On mémorise le dernier point d'ajout.
  finpour
  $\mathcal{F}$.**supprime**(f^α)
fintq
retourner $V^{\sim}_{somme\,profils}$

Algorithme 13 : L'algorithme de recomposition de la fonction de gain espéré $V^{\sim}_{somme\,profils}$

Ceci nous permet d'avoir très rapidement une sous-estimation de V^*, mais pas suffisante pour prendre de bonnes décisions dans le PRU$_0$ courante. C'est pour pour cette raison que nous avons affiné la méthode de décomposition/recomposition de V.

8.4 Amélioration de la méthode de décomposition/recomposition de la fonction $V^{\sim}$

En observant les profils de performance, nous avons constaté que les aspects de ces fonctions étaient différents et que pour certaines d'entre elles, les améliorations (relativement aux ressources dépensées) les plus caractéristiques se situaient en début de fonctions, alors que pour d'autres

l'amélioration était quasi constante. Pour la fonction f^{α} de la figure 8.8 par exemple, il existe un montant de ressource au delà duquel l'amélioration du gain espéré local ne sera pas vraiment significatif, puisque la courbe est quasi plate. Autrement dit, allouer plus de ressources que ce montant $\mathtt{r}$ situé juste avant le palier ne permet pas d'améliorer significativement cette tâche.

L'idée de la **deuxième méthode** est la suivante : quand l'agent a peu de ressources, il essaye de faire au moins une partie des tâches qui lui rapporteront le plus. S'il lui reste un peu de ressources, il essayera d'améliorer ces mêmes tâches avec le surplus. Ainsi, en repérant les profils de performance qui fournissent un bon rapport qualité/prix, nous saurons répartir nos ressources sur l'ensemble des tâches de la mission de façon à maximiser le gain espéré.

8.4.1 Décomposition fine des profils de performance

Nous avons donc décidé de séparer les fonctions de profil de performance en morceaux de façon à repérer les endroits où le rapport gain espéré, ressources consommées était maximum. Les ressources ne vont plus être allouées globalement mais petit à petit lors de la recomposition. Ceci va nous permettre de mieux adapter la fonction recomposée à la convexité globale de V^{*}, en mettant les meilleures améliorations en premier et les améliorations les plus faibles en dernier. Nous appelons rapport qualité/prix le rapport entre le gain espéré localement et les ressources nécessaires pour obtenir ce gain. Le but de notre méthode est d'allouer en priorité des ressources aux tâches qui fournissent le meilleur rendement. La recherche du meilleur rapport qualité/prix continue récursivement, jusqu'à ce que tous les profils soient traités.

Nous morcelons les profils de performance en deux étapes : nous recherchons les points qui fournissent le meilleur rapport qualité/prix puis nous découpons ces fonctions en deux parties : la partie à gauche du point de meilleur rapport sera notée g^{α} et le reste (à droite) du profil f_n^{α} sera redécoupé selon le même principe jusqu'à ce que la partie droite de la fonctions f_n^{α} soit vide. Les fonctions g^{α} sont stockées au fur et à mesure dans une liste triée par ordre croissant de rapport qualité/prix, pour que nous puissions donner la priorité aux tâches qui fournissent le meilleur rendement. Le morcellement est illustré sur la figure 8.10.

Les maxima de rapport qualité/prix sont sélectionnés en utilisant les profils de performances. Nous trouvons le premier avec la formule suivante :

$$\mathtt{r}^{max}_{\mathtt{p}^{\alpha},1} = \underset{\alpha,\mathtt{r}}{\operatorname{argmax}} \left(\frac{f^{\alpha}(\mathtt{r})}{\mathtt{r}}\right) \tag{8.5}$$

$$\forall i > 1, \mathtt{r}^{max}_{\mathtt{p}^{\alpha},i} = \underset{\alpha,\mathtt{r}}{\operatorname{argmax}} \left(\frac{f_i^{\alpha}(\mathtt{r})}{\mathtt{r}}\right) \tag{8.6}$$

Une fois le meilleur rapport qualité/prix sélectionné, nous décalons la courbe de façon à pouvoir y trouver un second meilleur rapport qualité/prix. Les équations suivantes décrivent le processus récursif de découpage des fonctions de profil de performance.

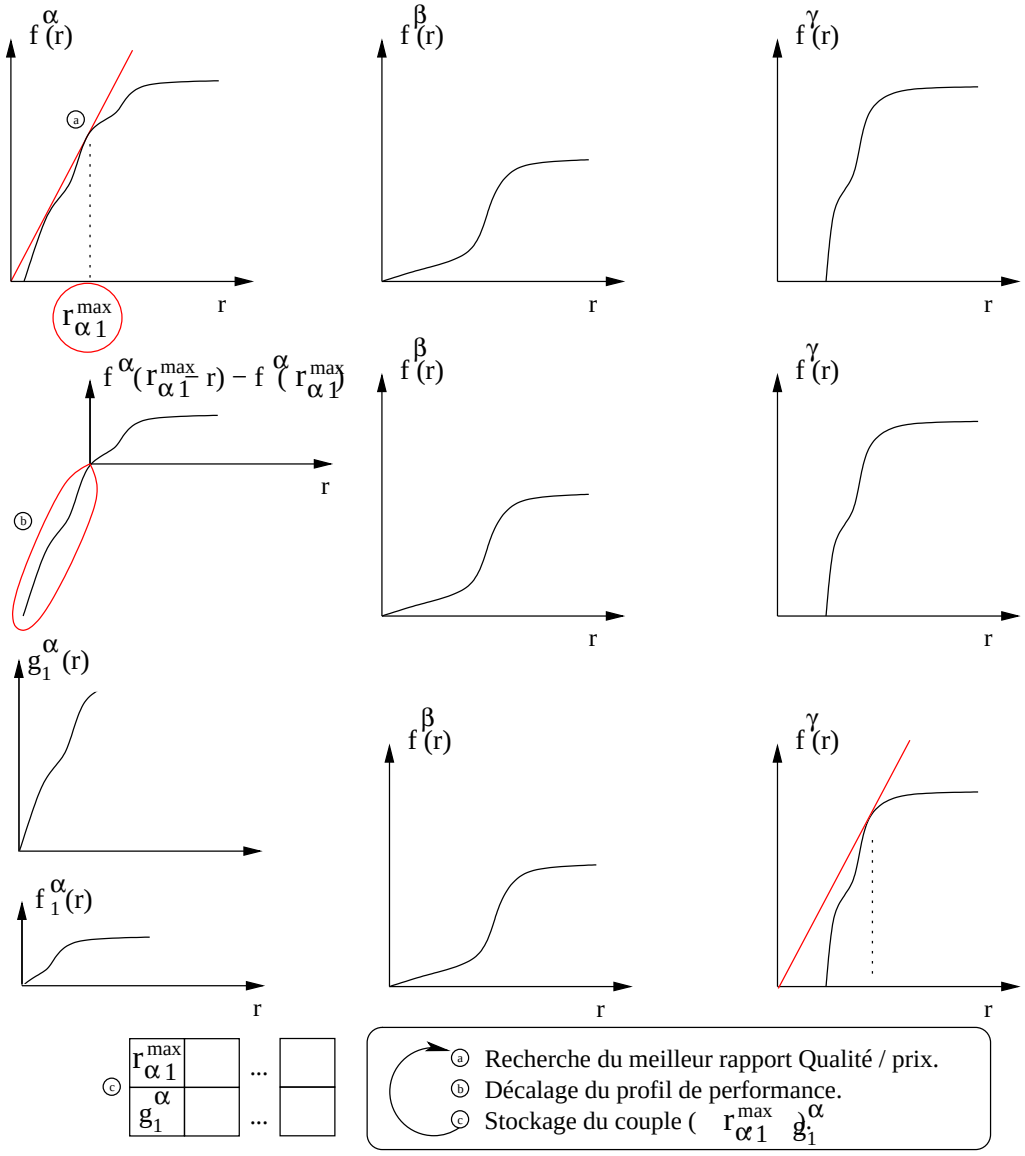


FIG. 8.10 – Découpage des profils de performance selon le meilleur rapport qualité/prix.

Explications *pour la figure 8.10 : nous sélectionnons à l'étape* **a** *le point de meilleur rapport (équation 8.5), en le sélectionnant parmi les types de PRUs* (α, β, γ)*. Sur notre figure, le meilleur rapport correspond au point* $\mathtt{r}_{\alpha,1}^{max}$*. Une fois le premier couple trouvé, nous passons à l'étape* **b** *: nous découpons le profil de performance en deux parties, la première partie* g^{α} *sera le premier morceau de courbe stocké (étape* **c***) et nous décalons l'origine du profil de performance jusqu'au point trouvé (équation 8.7) . Nous itérons ensuite ce processus, jusqu'à ce que tous les profils de performances soient traités. Sur notre exemple, le deuxième meilleur rapport se trouve sur le profil de performance de type* γ*.*

$$g_1^{\alpha} : \left| \begin{array}{lll} [0 \dots \mathtt{r}_{\mathtt{p}^{\alpha},1}^{max}] & \rightarrow & \mathbb{R} \\ \mathtt{r} & \rightarrow & f^{\alpha}(\mathtt{r}) \end{array} \right. \tag{8.7}$$

$$f_1^{\alpha} : \left| \begin{array}{lll} [0 \dots \mathtt{r}_{\mathtt{p}^{\alpha}}^{max} - \mathtt{r}_{\mathtt{p}^{\alpha},1}^{max}] & \rightarrow & \mathbb{R} \\ \mathtt{r} & \rightarrow & f^{\alpha}(\mathtt{r} - \mathtt{r}_{\mathtt{p}^{\alpha},1}^{max}) - f^{\alpha}(\mathtt{r}_{\mathtt{p}^{\alpha},1}^{max}) \end{array} \right. \tag{8.8}$$

$$\forall i > 1, \quad g_i^{\alpha} : \left| \begin{array}{lll} [0 \dots \mathtt{r}_{\mathtt{p}^{\alpha},i}^{max}] & \rightarrow & \mathbb{R} \\ \mathtt{r} & \rightarrow & f_i^{\alpha}(\mathtt{r}) \end{array} \right. \tag{8.9}$$

$$\forall i \geq 1, \quad f_{i+1}^{\alpha} : \left| \begin{array}{lll} \left[0 \dots \mathtt{r}_{\mathtt{p}^{\alpha}}^{max} - \sum_{j=1}^{i} \mathtt{r}_{\mathtt{p}^{\alpha},j}^{max}\right] & \rightarrow & \mathbb{R} \\ \mathtt{r} & \rightarrow & f_i^{\alpha}(\mathtt{r} - \mathtt{r}_{\mathtt{p}^{\alpha},i}^{max}) - f_i^{\alpha}(\mathtt{r}_{\mathtt{p}^{\alpha},i}^{max}) \end{array} \right. \tag{8.10}$$

Nous itérons ce processus jusqu'à ce que toutes les fonctions f^{α} soient entièrement morcelées et chaque morceau g^{α} soit trié dans une liste. Nous achevons ainsi la préparation nécessaire à la recomposition de la fonction de gain espéré si un imprévu vient modifier la séquence de tâches qui était prévue pour la mission.

Données : mission = $[\mathsf{PRU}_1, \dots \mathsf{PRU}_{\mathbf{P}}]$,$\mathcal{F}$
$TypeMission = \{\alpha, \text{tel que }, PRU^{\alpha} \in mission\}$
$\mathcal{G} = \emptyset$
pour $\alpha \in TypeMission$ **faire**
  tant que $f^{\alpha} \neq \emptyset$ **faire**
    $\mathtt{r}_{\mathtt{p}^{\alpha}}^{max} = \underset{\alpha,\mathtt{r}}{\text{argmax}}(f^{\alpha}(\mathtt{r})/\mathtt{r}, f^{\alpha} \in \mathcal{F})$
    $\forall \mathtt{r} \in [0, \dots, \mathtt{r}_{\mathtt{p}^{\alpha}}^{max}], g^{\alpha}(\mathtt{r}) = f^{\alpha}(\mathtt{r})$
    $\mathcal{G}$.**ajouter**$(\mathtt{r}_{max}^{\alpha}, g^{\alpha})$
    $f^{\alpha}(\mathtt{r}) = f^{\alpha}(\mathtt{r} - \mathtt{r}_{\mathtt{p}^{\alpha}}^{max}) - f^{\alpha}(\mathtt{r}_{\mathtt{p}^{\alpha}}^{max})$
  fintq
finpour
$\mathcal{G}$.**trier**$(g^{\alpha}(\mathtt{r}_{max}^{\alpha})/\mathtt{r}_{max}^{\alpha})$
retourner $\mathcal{G}$

Algorithme 14 : L'algorithme de découpage de profils de performance

Remarque : l'algorithme 14 consiste à découper tous les profils indépendamment les uns des autres (toujours selon le critère de rapport qualité/prix), puis à stocker les couples $(\mathtt{r}_{\mathtt{p}^{\alpha},i}^{max}, g_i^{\alpha})$ dans une liste que nous retrions à la fin de l'algorithme pour avoir en premier le meilleur rapport qualité/prix du meilleur type de PRU.

8.4.2 Recomposition de la fonction de gain espéré $V^{\sim}_{morceauxprofils}$

L'algorithme 15 de recomposition de la fonction de valeur est identique à l'algorithme 13, à ceci près que les données sont différentes : il suffit de remplacer les profils de performances f^{α} par les morceaux de profils de performances $\mathcal{G} = \{g_i^{\alpha}\}$. Le tri des morceaux de profil de performance g_i^{α} vient d'être expliqué dans la sous-section précédente. Nous n'avons plus qu'à recoller dans l'ordre les morceaux de courbes g_i^{α} pour obtenir la fonction de gain espéré $V^{\sim}_{morceaux\,profils}$ que nous appelons la fonction de gain espéré **recomposée par ajout de morceaux de profils de performance**. La figure 8.11 illustre la construction de cette fonction.

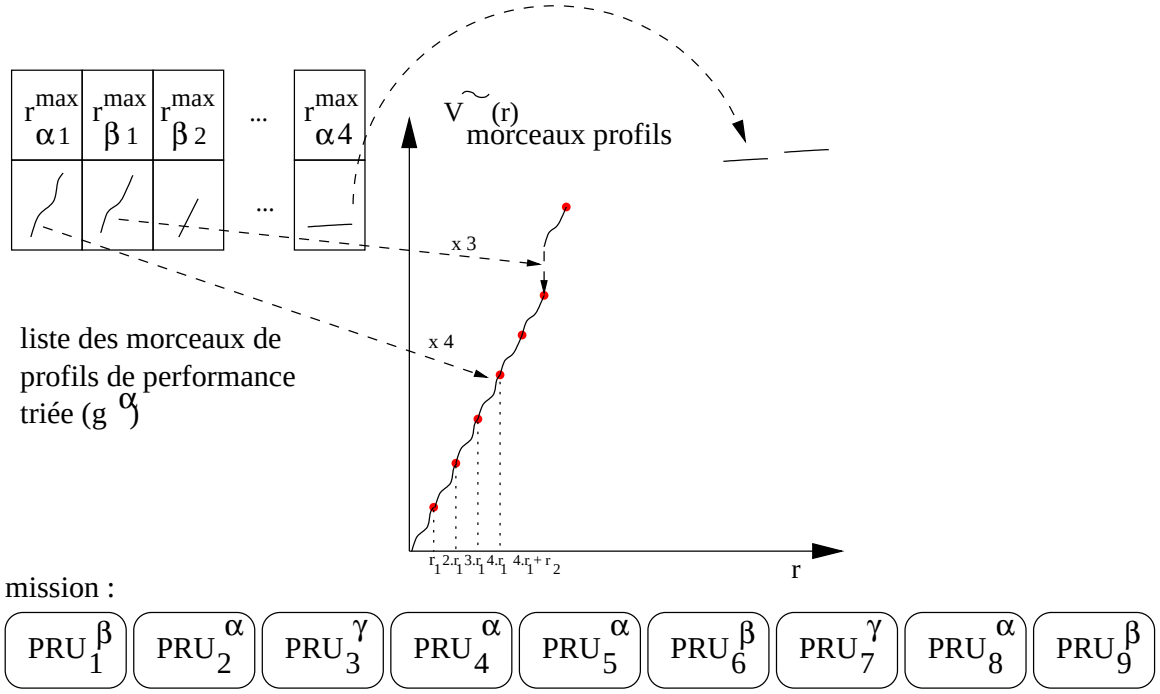


FIG. 8.11 – Recomposition de la fonction de gain espéré $V^{\sim}_{morceauxprofils}$.

Explications *pour la figure 8.11 : cette figure explique comment approximer par recomposition la fonction de gain espéré. Dans cet exemple, nous avons une mission composée de* 4 *PRUs de type* α *et* 3 *PRUs de type* β *dans la séquence de tâches future. Comme l'indique la liste, le premier maximum de rapport qualité prix avait été trouvé pour la* PRU^{α}*. Nous rajoutons donc* 4 *fois le morceau de courbe correspondant à cette tâche* (g_1^{α})*, puis trois fois le morceau de courbe correspondant à la deuxième tâche*(g_1^{β})*, jusqu'à arriver au montant de ressource courant sur l'axe des abscisses.*

L'algorithme pour obtenir $V^{\sim}_{morceaux\,profils}$ demande un temps de calcul très inférieur à la méthode pour obtenir l'optimal V^* puisque nous ne faisons que quelques additions de morceaux de courbes, alors que l'évaluation par programmation dynamique de la fonction de gain espéré

optimal nécessite un calcul qui évalue chaque état de l'ensemble des PRUs qui constituent la suite de la mission. Sa complexité est de l'ordre du nombre de ressources disponibles, soit $\mathcal{O}(\mathtt{R})$, puisque nous construisons cette courbe point par point.

Données : mission = $[\mathsf{PRU}_1, \dots \mathsf{PRU}_{\mathbf{P}}]$, $\mathcal{G}$
$\mathtt{r}_{allouee} = 0$ //ressource déjà allouée pour la mission.
$V_{prec} = 0$ dernière valeur de la courbe.
tant que $\mathtt{r}_{allouee} <= \mathtt{r}^{max}_{mission}$ **faire**
  $(\alpha, i) = \underset{(\alpha,i)}{\operatorname{argmax}} \left(g_i^{\alpha}(\mathtt{r}^{max}_{\mathtt{p}^{\alpha},i}), g_i^{\alpha} \in \mathcal{G}\right)$ type de la meilleure PRU
  $n_{\alpha} = \mathbf{nb}(\mathsf{PRU}^{\alpha} \in mission)$ nombre de PRU^{α} dans la mission
  pour $i \in [1, \dots, n_{\alpha}]$ **faire**
    pour $\mathtt{r} \in [\mathtt{r}_{allouee}, \dots, \mathtt{r}_{allouee} + \mathtt{r}^{max}_{\mathtt{p}^{\alpha},i}]$ **faire**
      $V^{\sim}_{sommeprofils}(\mathtt{r}) = V_{prec} + g_i^{\alpha}(\mathtt{r})$ Copie des valeurs du meilleur profil actuel
    finpour
    $\mathtt{r}_{allouee} += \mathtt{r}^{max}_{\mathtt{p}^{\alpha},i}$
    $V_{prec} = V^{\sim}_{morceaux\,profils}(\mathtt{r}_{allouee})$ On mémorise le dernier point d'ajout.
  finpour
  $\mathcal{G}$.**supprime**(g_i^{α})
fintq
retourner $V^{\sim}_{morceaux\,profils}$

Algorithme 15 : L'algorithme de recomposition de la fonction $V^{\sim}_{morceaux\,profils}$

Nous avons également calculé sur le même principe une fonction recomposée par ajout de morceaux linéaires, qui sont les morceaux de profils de performance que nous linéarisons. Les fonctions g_i^{α} sont transformées en droites l_i^{α} par la formule suivante 8.11,

$$\forall \alpha, \forall i, \quad l_i^{\alpha} : \left| \begin{array}{lcl} [0 \dots \mathtt{r}^{max}_{\mathtt{p}^{\alpha},i}] & \rightarrow & \mathbb{R} \\ \mathtt{r} & \rightarrow & \frac{\mathtt{r}}{\mathtt{r}^{max}_{\mathtt{p}^{\alpha},i}} . g_i^{\alpha}(\mathtt{r}^{max}_{\mathtt{p}^{\alpha},i}) \end{array} \right. \tag{8.11}$$

De cette façon, la fonction de gain espéré approchée obtenue[47] a une allure plus régulière puisque c'est une suite de segments plutôt qu'une suite de courbes. Nous l'appelons **la fonction d'approximation linéaire par morceaux** et nous la notons $V^{\sim}_{morceaux\,lineaire}$. Nous présentons finalement toutes les approximations obtenues sur la figure 8.12.

Sur la figure 8.12, nous avons trois approximations de la courbe de gain espéré : la courbe pleine représente le gain espéré optimal calculé en quelques dizaines de secondes voire minutes par programmation dynamique. Juste en dessous, nous avons la première approximation linéaire par

[47] La méthode de recomposition reste évidemment la même.

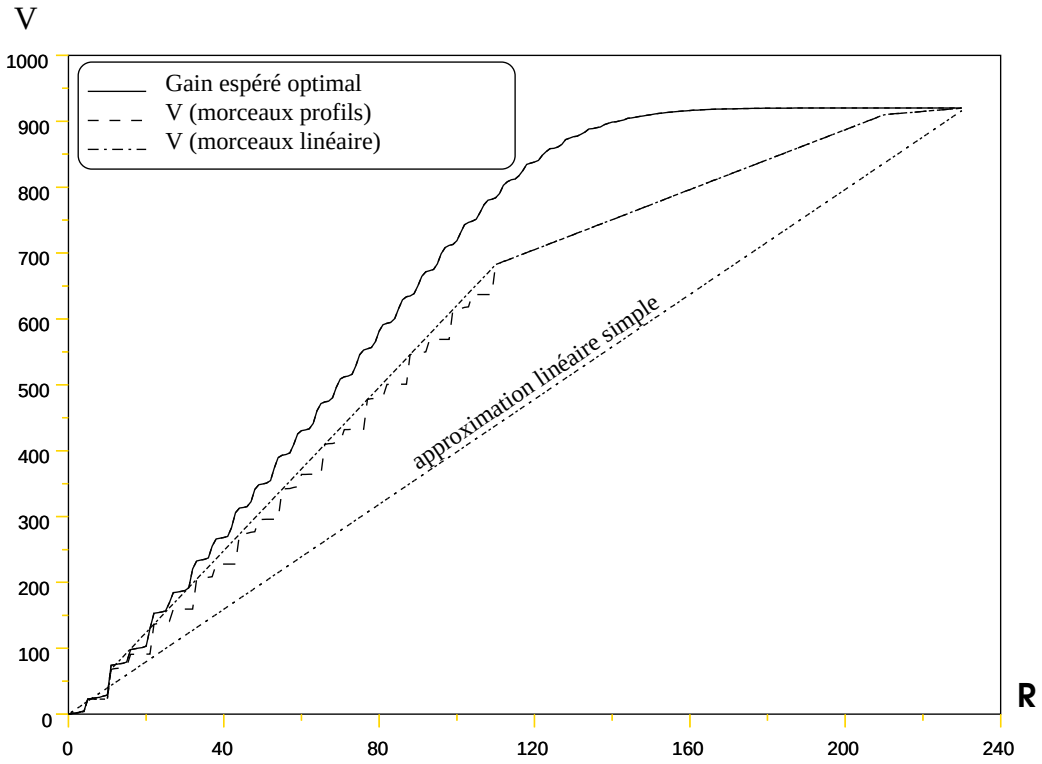


FIG. 8.12 – Approximations de la fonction V par recomposition.

morceaux $V^{\sim}_{morceaux\,lineaire}$*(en ..-..) et encore en dessous l'approximation par ajout de morceaux de profils* $V^{\sim}_{morceaux\,profils}$*. La dernière approximation est l'approximation linéaire* $V^{\sim}_{lineaire}$ *présentée en tout début de chapitre.*

8.4.3 Comparaison de la fonction recomposée $V^{\sim}$ avec V^{*}.

La fonction obtenue est bien entendu différente de la fonction optimale. Cependant, nous avons remarqué de choses intéressantes, que nous illustrons sur la figure 8.13.

Remarque 1 : Si la consommation de ressources de tous les modules est déterministe, la fonction de gain espéré obtenue par la méthode de recomposition par morceaux de courbes est égale en certains points à la fonction de gain espéré optimale.

Nous n'avons pas encore exploité cette propriété, mais nous avons l'intention d'utiliser prochainement cette propriété pour calculer une fonction de gain espéré optimiste approchée et une fonction de gain espéré pessimiste approchée. En effet, il suffit de considérer pour la fonction pessimiste que le robot va toujours consommer une quantité maximale de ressources et de ce fait, les PRUs deviendront artificiellement déterministes. Il suffira alors pour calculer la fonction de gain espéré pessimiste de faire une approximation en recomposant la fonction par ajout de morceaux de profils de performance obtenues avec les PRUs pessimistes déterministes. Cela nous

donnera une autre méthode pour obtenir une sous-estimation de la fonction de gain espéré.

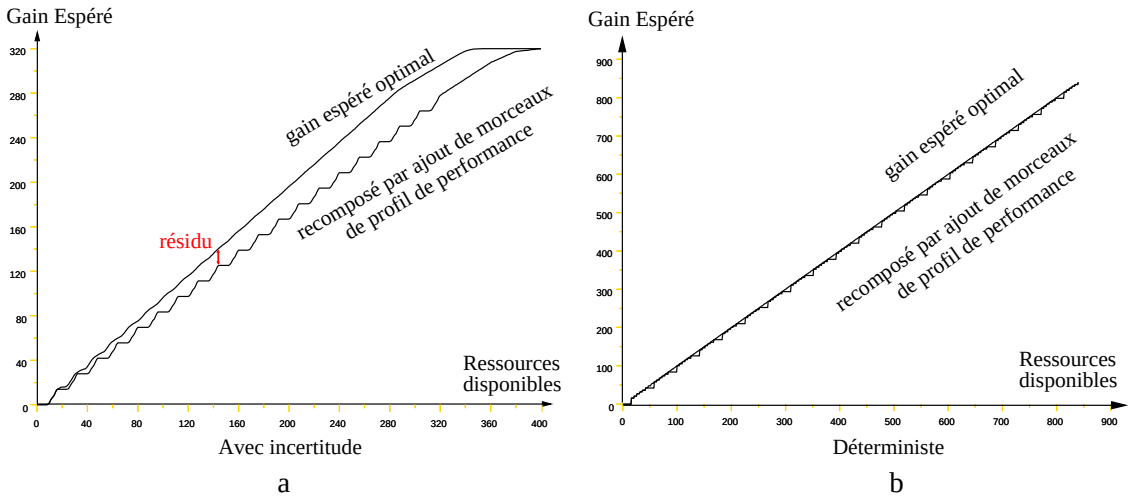


FIG. 8.13 – Observations à propos de la fonction de gain espéré recomposée.

Sur la figure 8.13, nous observons deux choses remarquables : sur la figure, on voit que la fonction de gain espéré obtenue par recomposition de morceaux de profils de performance sous-estime la fonction optimale. À droite, figure b, nous avons utilisé des PRUs dont les modules consomment une quantité déterminée de ressources. Le gain recomposé par ajout de morceaux de profils de performanceest égale la fonction optimale en un grand nombre de points.

Nous avons fait une autre remarque intéressante, valable cette fois-ci même quand la distribution de probabilités de consommation de ressources des modules est probabiliste :

Remarque 2 : Quel que soit le montant de ressources disponibles, la fonction de gain espéré obtenue par la méthode de recomposition par morceaux de courbes sous-estime la fonction de gain espéré optimale.

$$\forall \mathbf{r} \in \mathtt{R}, V^{\sim}_{morceaux\,profils}(\mathbf{r}) \quad \leq \quad V^{*}(\mathbf{r}) \tag{8.12}$$

Cette propriété devient fausse lorsque l'on utilise un approximation linéaire par morceaux $V^{\sim}_{morceaux\,lineaire}$.

Expliquons ce phénomène de sous-estimation : en allouant une certaine quantité de ressources à une PRU donnée, nous pouvons obtenir à la fin de celle-ci un résidu de ressources non utilisé. Le gain espéré local donné par un profil de performance ne nous indique que la valeur que nous pouvons espérer obtenir en sortie de PRUs, mais pas les ressources qui resteront. Il est possible en effet de consommer moins de ressources que ce qui est initialement alloué (mais jamais plus). L'agent peut de ce fait faire des économies de ressources. Il conserve alors un résidu de ressources

inutilisées. Ce phénomène se cumule sur l'ensemble des PRUs de la mission, comme le montre la figure 8.13 et cette répercussion éloigne de plus en plus $V^{\sim}_{morceaux\,profils}$ de V^{*}.

8.5 Dernier essai d'amélioration pour approcher V.

Nous verrons dans le chapitre 11 que l'utilsation de $V^{\sim}_{morceaux\,lineaire}$ permet de générer un comportement « acceptable » pour le robot en cas de changement dans la mission. Nous avons néamoins voulu savoir s'il était possible de continuer d'améliorer cette méthode en réutilisant dans le calcul les résidus de ressources potentiellement non consommé pendant la mission.

8.5.1 Détection de résidus de ressources non utilisés lors de l'allocation

La probabilité d'obtenir un résidu de ressources après l'exécution d'une PRU à laquelle nous allouons une quantité $\mathtt{r_d}$ de ressources est la probabilité de ne pas avoir tout dépensé lorsque l'agent termine le dernier niveau, c'est-à-dire la probabilité d'être dans un état $[\mathtt{r} > 0, \mathbf{Q}, \mathtt{p}, \mathtt{N_p}]$. Nous pouvons calculer cette probabilité par programmation dynamique. Il suffit pour cela de développer l'arbre correspondant à la politique locale optimale, c'est-à-dire le même arbre que celui qui va permettre de calculer le profil de performance pour cette PRU. Ensuite, nous allons conserver les états feuilles de cet arbre, puis remonter jusqu'à l'état racine $[\mathtt{r_d}, 0, \mathtt{p}, 0]$ en combinant les différentes probabilités de consommation de ressource. En procédant ainsi, nous obtenons un arbre réduit probabiliste qui nous indique la probabilité d'avoir $\mathtt{r}$ ressources restantes à la fin de l'exécution de la PRU, nous pouvons également connaître la probabilité de nous trouver dans un état d'échec. L'arbre réduit probabiliste peut être représenté par un réseau bayésien. Nous notons π^{*}_{α} la politique locale optimale d'une PRU_{α}. L'ensemble des feuilles d'un arbre résiduel pour un état racine $[\mathtt{r_d}, 0, \mathtt{p}, 0]$ donné est l'ensemble des états du dernier niveau de la PRU^{α} qui sont accessibles depuis la racine en appliquant les actions qui sont dictées par la politique π^{*}_{α}, ainsi que l'état d'échec. L'arbre résiduel peut se calculer simplement en trois étapes :

1. calcul de la politique π^{*}_{α},
2. recherche des états feuilles accessibles,
3. combinaison des probabilités des niveaux intermédiaires par chaînage arrière pour connaître la probabilité d'accéder à une des feuilles en suivant π^{*}_{α}.

Ce mécanisme est illustré sur la figure 8.14.

8.5.2 Ré-injection des résidus de ressources non utilisés lors de l'allocation

Après avoir calculé les arbres de résidus de ressources (ce calcul sera fait hors ligne), nous présentons ici le mécanisme de reconstruction de la fonction de valeur avec les résidus que nous appelons $V^{\sim}_{residus}$. Cette construction se base sur la construction par morceaux de profils : nous

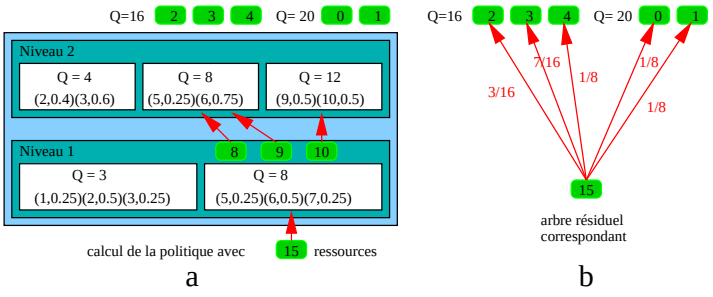


FIG. 8.14 – Arbre résiduel de ressources

Explications *pour la figure 8.14 : nous représentons sur la figure 8.14.a une PRU composée de deux niveaux. Chaque module contient une qualité et une distribution de probabilités. Nous commençons l'exécution de cette PRU avec* 15 *unités de ressource. La politique optimale pour une mission composée d'une seule PRU de ce type est d'exécuter le deuxième module du premier niveau, et selon qu'il reste plus ou moins de* 9 *unités de ressource, le troisième ou le deuxième module du dernier niveau. Donc en suivant la politique optimale locale à cette PRU avec* 15 *unités de ressource, il restera entre* 0 *et* 4 *unités de ressource. Nous modélisons ceci sur la figure 8.14.b. Les probabilités associées dépendent de la distribution de probabilités des modules de la PRU.*

mettons encore bout à bout les morceaux de profils de performances g, mais nous essayons de rejoindre la courbe optimale en ajoutant par endroit la valeur qui pourrait être obtenue en utilisant les résidus de ressources non consommés. Schématiquement, nous essayons de combler les espaces résiduels de la figure 8.13.a.

Notre idée est de rajouter après chaque ajout de morceau de courbe la valeur qui pourrait être obtenue avec le résidu de ressources. Comme nous avons plusieurs quantités de résidus possibles, nous pondérons cet ajout par les probabilités de l'arbre résiduel correspondant.

Par exemple, considérons une mission de 3 PRUs du type de la figure 8.14. L'agent dispose de 30 unités de ressources au départ. Nous avons g_1^α et $\mathtt{r}_{\mathtt{p}^\alpha,1}^{max} = 15$ en première position de $\mathcal{G}$. Nous allouons alors 15 unités de ressources à la première PRU et ensuite nous n'allouons plus seulement $15 = 30 - 15$ ressources à la deuxième PRU, mais 15, 16, 17, 18 ou 19 ressources avec les probabilités respectives de $1/8, 1/8, 3/16, 7/16, 1/8$ (conformément à ce qu'indique la figure 8.14.b). La méthode pour obtenir $V^{\sim}_{morceaux\,profils}$ consiste à ajouter un à un tous les morceaux de courbes de $\mathcal{G}$. Nous adaptons cette même méthode en ré-injectant les résidus et nous appelons notre nouvelle approximation $V^{\sim}_{residus}$. Sa construction se fait en deux temps : nous commençons par mettre à plat dans l'ordre tous les morceaux de profils de performance ordonnés (comme pour la méthode précédente mais sans ajout) comme l'indique l'algorithme 16, puis nous ajoutons tous ces morceaux en tenant compte des résidus, comme le montre l'algorithme 17.

Dans un premier temps, nous mettons tous les profils de performance à plat et dans l'ordre.

Données : mission = $[\mathsf{PRU}_1, \ldots \mathsf{PRU}_{\mathbf{P}}]$, $\mathcal{G}$
$\mathtt{r}_{allouee} = 0$ ressources déjà allouées pour la mission.
tant que $\mathtt{r}_{allouee} <= \mathtt{r}^{max}_{mission}$ **faire**
  $(\alpha, i) = \underset{(\alpha,i)}{\text{argmax}} \left(g_i^\alpha(\mathtt{r}^{max}_{\mathtt{p}^\alpha,i}), g_i^\alpha \in \mathcal{G}\right)$ type de la meilleure PRU
  $n_\alpha = \mathbf{nb}(\mathsf{PRU}^\alpha \in mission)$ nombre de PRU^α dans la mission
  pour $i \in [1, \ldots, n_\alpha]$ **faire**
    pour $\mathtt{r} \in [\mathtt{r}_{allouee}, \ldots, \mathtt{r}_{allouee} + \mathtt{r}^{max}_{\mathtt{p}^\alpha,i}]$ **faire**
      $V_{ordonne}(\mathtt{r}) = g_i^\alpha(\mathtt{r})$ Copie des valeurs du meilleur profil actuel
    finpour
    $\mathtt{r}_{allouee} += \mathtt{r}^{max}_{\mathtt{p}^\alpha,i}$
    si $i == 1$ **alors**
      $A_{\mathtt{r}_{allouee}} = \mathbf{arbreResiduel}(g_i^\alpha(\mathtt{r}^{max}_{\mathtt{p}^\alpha,i}))$ Le calcul est fait off-line
    sinon
      $A_{\mathtt{r}_{allouee}} = [(1, 0)]$ pour les morceaux de courbe qui ne correspondent pas à un début de courbe, nou négligeons le résidu
    finsi
  finpour
  $\mathcal{G}$.**supprime**(g_i^α)
fintq
retourner $V_{ordonne}$

Algorithme 16 : Mise en ordre des profils de performances sans cumul de valeur.

Chaque fois que nous finissons d'ajouter un profil de performance, nous enregistrons un arbre résiduel A_r à la fin du morceau de courbe g^α. Nous ne le faisons que pour les morceaux de courbes qui forment les premières parties du découpage (les g_1^α). Ensuite, il ne reste plus qu'à ajouter les morceaux mis à plat les uns par dessus les autres.

L'algorithme 17 consiste à mettre bout à bout toutes les courbes mises à plat précédemment dans $V_{ordonne}$. Pour cela, nous utilisons simplement un point de stockage V_{prec} qui stocke la valeur du dernier point non nul calculé. Les valeurs qui sont ajoutées ne sont plus les valeurs seulement des morceaux de courbes, mais une somme pondérée des valeurs de ces morceaux de courbes par les résidus (et leur probabilité) des arbres précédemment construits. Au départ, il n'y a pas de résidu, donc nous avons une probabilité 1 d'avoir 0 en résidu. Ensuite, nous changeons d'arbre des résidus chaque fois qu'il existe un nouvel A_r non vide. De cette façon, nous rehaussons légèrement la valeur du gain espéré par rapport à $V^{\sim}_{morceaux\,profils}$.

Cette méthode a l'avantage de se rapprocher davantage de la fonction de gain espéré optimale, du moins en début de courbe. Mais il y a plusieurs inconvénients à cette dernière méthode : elle

Données : mission = $[\mathsf{PRU}_1, \ldots \mathsf{PRU}_{\mathbf{P}}]$, $V_{ordonne}$

$V_{prec} = 0$ dernière valeur de la courbe.

$residus = [(1, 0)]$ liste de probabilité, ressources restantes.

pour $0 \leq \mathtt{r} < \mathtt{r}^{max}_{mission}$ **faire**

  $V^{\sim}_{residus}(\mathtt{r}) = V_{prec} + \sum_{(p,\mathtt{r}') \in residus} p.V_{ordonne}(\mathtt{r} + \mathtt{r}')$

  si $V_{ordonne}(\mathtt{r} + 1) == 0$ **alors**

    $V_{prec} = V^{\sim}_{residus}(\mathtt{r})$ On mémorise le dernier point d'ajout.

  finsi

  si $A_{\mathtt{r}} \neq \emptyset$ **alors**

    $residus = A_{\mathtt{r}}$

  finsi

finpour

retourner $V^{\sim}_{residus}$

Algorithme 17 : L'algorithme de recomposition de la fonction $V^{\sim}_{residus}$

est plus longue à calculer que le simple ajout de morceaux de profils et les résidus ré-injectés génèrent des résidus de résidus. Nous pourrions essayer de ré-injecter les résidus de résidus et essayer de converger de fil en aiguille vers la fonction optimale, mais le temps nécessaire est finalement du même ordre que celui qui permet d'obtenir la fonction optimale. C'est pourquoi nous avons abandonné pour l'instant l'idée de ré-injecter les résidus pour rétablir la pente de la fonction de valeur approchée.

Nous verrons également dans le chapitre 11 que les méthodes $V^{\sim}_{morceaux\,profils}$ et $V^{\sim}_{morceaux\,lineaire}$ approchent suffisamment la fonction de gain espéré optimale pour ne pas générer trop d'erreurs au niveau de la politique locale.

La méthode d'ajout de résidus est finalement :

- plus coûteuse en temps de calcul que l'ajout de morceaux de profils,
- n'améliore pratiquement pas la qualité des décisions prises par le robot qui suit une politique basée sur $V^{\sim}_{morceauxprofils}$ ou $V^{\sim}_{morceauxlineaire}$.

Le chapitre 11 montre expérimentalement que la méthode sans ré-injection de résidus suffit pour obtenir un comportement qui s'approche de l'optimal. Pour cette raison, nous avons conclu qu'il n'était pas forcément nécessaire ni judicieux de continuer à améliorer notre méthode, ni d'exposer les performances de la méthode avec résidus. Le calcul de $V^{\sim}_{morceauxprofils}$ ou $V^{\sim}_{morceauxlinaire}$ suffit pour que le robot puisse contrôler sa consommation de ressource en cas de changement de mission.

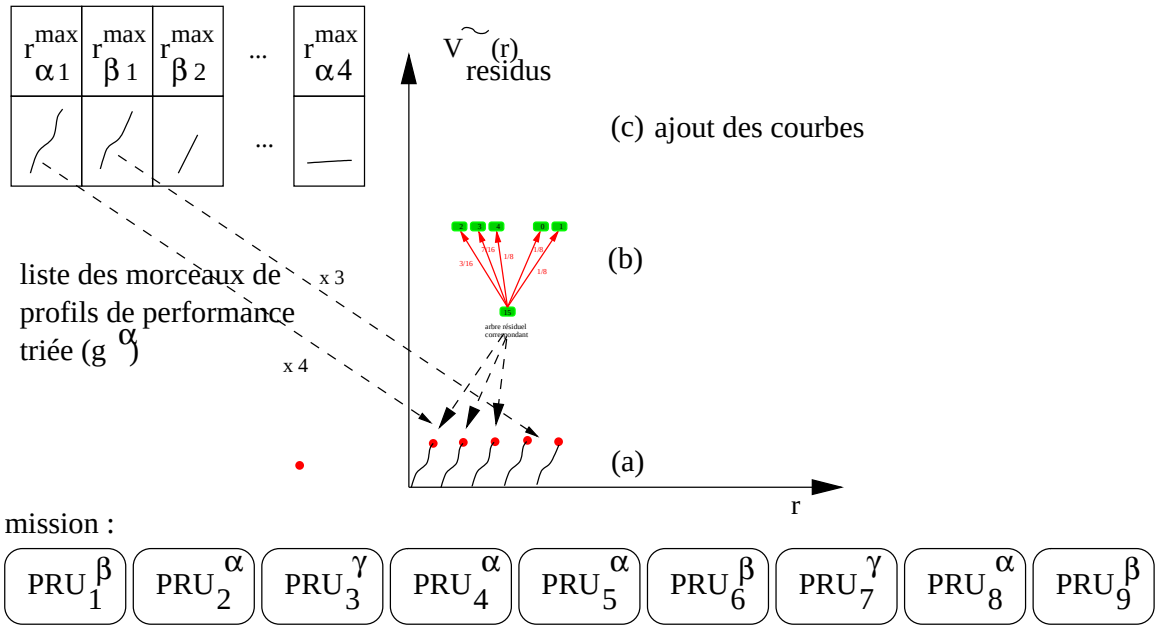


FIG. 8.15 – Construction de $V_{residus}$.

Explications *pour la figure 8.15 : la construction de la fonction en tenant compte des résidus se fait en deux étapes : nous alignons les morceaux dans l'ordre, puis nous les ajoutons bout à bout en injectant cette fois les résidus de ressources. En (a), nous mettons les profils bout à bout, puis nous ajoutons à chaque bout de courbe un arbre résiduel (b) (calculé avant la mission). Ensuite, nous appliquons l'algorithme 17 (c) pour ajouter les morceaux de courbes de (a) combinés avec les arbres de (b).*

Conclusion

Dans le chapitre précédent, nous avons présenté l'environnement dynamique dans lequel pourrait évoluer notre robot et une structure doublement adaptative permettant de tenir compte à la fois des incertitudes de consommation de ressources mais aussi d'éventuels changements imprévus pendant la mission. Ce chapitre constitue le cœur de notre deuxième apport. Nous proposons une méthode permettant à un robot de calculer immédiatement une politique dès qu'un changement intervient dans la séquence de tâche à venir. Cette politique s'applique à la PRU courante, elle n'est pas forcément optimale. Mais elle permet au robot de prendre de bonnes décisions, en attendant de pouvoir recalculer une politique optimale.

Le calcul de la politique locale se fait en deux temps : le robot commence par calculer le gain espéré pour la séquence de tâches à venir en fonction des ressources qu'il lui restera, puis, en fonction de cela, il calcule une politique pour la PRU qu'il va ou qu'il est en train d'exécuter. Le calcul d'une politique pour une seule PRU est immédiat.

Pour obtenir immédiatement une politique, nous proposons donc d'approcher la fonction de gain espéré définie sur les nouvelles tâches après le changement qui est de forme convexe. Pour ceci, nous avons mis au point plusieurs algorithmes qui permettent de reconstruire par ajouts successifs de morceaux de fonction de gain espéré locaux une fonction de gain espéré globale approchée. Nous avons principalement retenu deux de ces méthodes $V^{\sim}_{morceaux\,profils}$ et $V^{\sim}_{morceaux\,lineaire}$. Nous observons que cette fonction approchée est une sous-estimation de la fonction de gain espéré obtenue par une méthode de calcul optimale.

Nous verrons dans le chapitre 11 au travers d'expériences, que non seulement cet algorithme fournit immédiatement une fonction de gain espéré, mais aussi que la qualité des décisions prises par le robot qui suit une politique obtenue par le biais d'une fonction de gain espéré approchée est bonne.

Conclusion de la troisième partie

Nous avons présenté dans cette partie un système de contrôle doublement adaptatif pour un robot autonome qui doit exécuter une mission. Le robot doit adapter d'une part sa consommation de ressources en allouant à chaque tâche de la mission une partie de celles-ci. Il doit d'autre part s'adapter à un éventuel changement de mission pendant cette mission.

Notre agent rationnel maximise tout au long de sa mission un critère sur lequel il basera son comportement. Celui-ci dépend de la façon dont il va effectuer les tâches qu'on lui propose : l'agent reçoit à la fin de chaque PRU une récompense en relation avec la qualité du résultat qu'il aura produit.

Nous avons tiré parti du raisonnement progressif, présenté dans les chapitres 4 et 5, pour modéliser les tâches qui compose cette mission. Nous avons déjà vu dans ces chapitres qu'il était possible d'obtenir un contrôle optimal global qui s'adapte à la consommation incertaine de ressource et d'avoir de la part du robot un comportement rationnel visant à maximiser la somme des qualités accumulées dans les tâches modélisées sous formes de unités de raisonnement progressif tout en gérant au mieux les ressources restantes dans le robot.

Pour obtenir ce comportement, il faut calculer une politique. Les algorithmes de programmation dynamique classiques permettent de trouver une politique optimale, mais le temps nécessaire pour l'obtenir augmente considérablement dès que le nombre de PRUs devient élevé.

L'adaptation à un changement de mission pendant l'exécution de celle-ci n'est pas envisageable avec un tel système, puisque le temps nécessaire pour retrouver un comportement optimal peut faire perdre un temps précieux à notre robot. Dès lors, nous avons envisagé de calculer une politique non plus optimale mais bonne, de façon quasi instantanée, pour que le robot puisse adopter un comportement satisfaisant dès qu'un changement intervient.

Ce calcul est fondé sur une évaluation instantanée du gain espéré : en séparant le calcul du gain espéré (futur) de la politique à appliquer localement (présent), nous pouvons envisager d'obtenir la politque rapidement, sous réserve de connaître ce gain espéré.

Le calcul instantané de la fonction de gain espéré est inspiré d'un mécanisme de décomposition/recomposition de MDP de grandes tailles : nous calculons pour chaque tâche un profil de performance indépendant de la mission. Ces calculs sont faits en créant pour chaque tâche des

missions abstraites composées d'une unique et seule tâche. Le profil de performance d'une tâche est le gain espéré de chacune de ces missions abstraites. Ensuite, les profils de performances sont découpés en morceaux puis triés. La recomposition de la fonction de valeur consiste alors à mettre bout à bout ces morceaux de profils de performances triés, de façon à obtenir une fonction de gain espéré approchée qui sous-estime la fonction de gain espéré optimale que nous n'avons pas le temps de calculer. Finalement, nous calculons une politique locale en fonction de ce gain espéré approché et le robot adopte le comportement indiqué par cette politique.

Nous avons établi un protocole expérimental dans le chapitre 11 pour valider notre approche.

Quatrième partie

Validation des résultats

Chapitre 9

Validation : mise en œuvre du raisonnement progressif sur un robot

Introduction

Notre toute première idée dans cette thèse est d'appliquer le raisonnement progressif au contrôle d'une mission effectuée par un robot autonome en environnement réel. Nous voulons démontrer qu'avec un tel mécanisme, le robot est capable d'adapter sa décision en fonction des ressources qu'il lui reste. Nous nous sommes donc servis d'un robot Koala, acquisition récente de l'équipe MAD pour mettre en œuvre le scénario du robot bowling. La mission consiste à faire tomber des quilles sur différents sites pour marquer le plus de points possible. Les quilles sont posées « à la main » sur le sol, introduisant un facteur d'incertitude pour les détecter et pour les faire tomber.

9.1 Le robot Koala

Le robot Koala provient de l'entreprise K-TEAM http://www.k-team.com/. C'est un robot de taille moyenne conçu pour des applications en environnement réel.

9.1.1 Caractéristiques

Il est plus grand et plus puissant que le Khepera (K-TEAM), et on peut lui rajouter de nombreux accessoires. Le Koala dont dispose l'équipe est muni d'une caméra[48]. Il se déplace grâce à six roues, et peut s'adapter à tous types de terrain en intérieur. Ses six roues lui permettent d'ailleurs de tourner sur lui-même (voir figure 9.1.1). En faisant avancer les roues du côté gauche et reculer les roues du côté droit à la même vitesse, il effectue une rotation sur la droite. Tout

[48] Malheureusement, la carte d'acquisition n'est pas fournie avec la caméra.

autour du robot Koala sont disposés des capteurs infrarouges qui lui permettent de détecter d'éventuels obstacles.

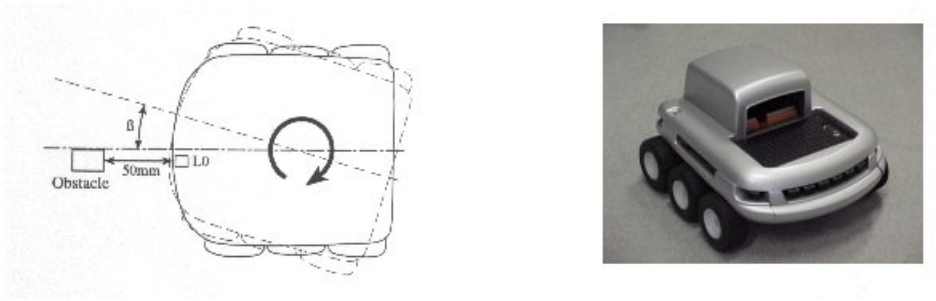


FIG. 9.1 – Le robot Koala.

De plus, il est dit sur le site de K-Team que le BIOS de tous les robots de la marque sont compatibles (le Khepera et le Koala). Le robot Koala peut être programmé via différents environnement de développement comme LabVIEW® , MATLAB® ou SysQuake. Nous avons opté pour un développement en C ; un outil de cross-compilation est fourni avec le robot. Le transfert des données et des programmes se fait par l'intermédiaire d'un port série standard, en branchant un câble depuis le PC vers le robot. Le système de gestion de l'énergie est performant, et les batteries ont une capacité et une autonomie largement suffisante pour les expériences que nous avons menées.

9.1.2 Comment profiter du robot Koala ?

Le robot Koala est donc un outil performant pour effectuer des expériences robotiques en intérieur. Seulement, certains avantages peuvent parfois devenir des inconvénients. Nous aurions voulu montrer par exemple qu'il était possible de gérer deux ressources consommables différentes à la fois. Mais comme le système de gestion de l'énergie est performant, cela n'a pas été possible.

Nous avons alors réfléchi à d'autres possibilités de ressources consommables : un bac contenant de l'eau ou du sable par exemple aurait pu servir de ressource consommable. Si nous avions eu un tel dispositif, nous aurions pu mettre en place une mission qui consiste à arroser plusieurs sites (des fleurs par exemple), en fonction du taux d'humidité. Cependant, nous de disposons pas des moyens techniques suffisants pour concevoir un tel dispositif. Pour toutes ces raisons, les expériences que nous avons menées ne traitent qu'une seule ressource consommable : le temps.

9.1.3 Détecter puis renverser une quille

Le robot Koala ne possède qu'un seul type de capteur. Les capteurs infrarouges sont disposés à intervalles réguliers autour de celui-ci, ils permettent de détecter précisément un obstacle à moins

de 50 centimètres de distance. Pour utiliser ces capteurs, nous avons choisi de faire détecter des objets au robot. Grâce à deux gobelets posés l'un sur l'autre, nous avons créé une quille, au dessus de laquelle nous avons placé des balles de couleur. Les couleurs sont justes présentes pour nous aider, le robot ne détecte pas les couleurs. Pour détecter une quille, nous utilisons un seul des capteurs avant du robot. Le robot tourne sur lui-même, puis dès que le rayon est coupé, il sait qu'une quille lui fait face (voir figure 9.2.b). Les quilles sont posées loin des murs, pour éviter les confusions lors de la détection. Si le robot doit détecter plusieurs quilles, nous devons les espacer suffisamment pour que celui-ci puisse les distinguer.

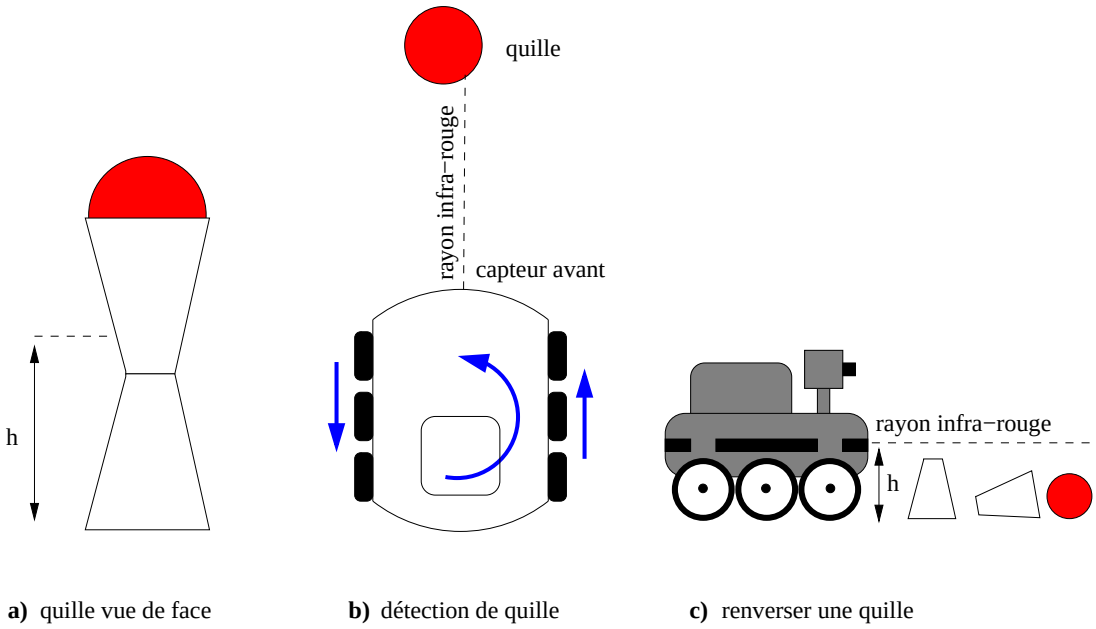


FIG. 9.2 – Comment détecter puis renverser une quille.

Pour renverser une quille, le robot avance sur une longueur égale à sa portée de détection maximale (soit 50 centimètres). De cette façon, il est certain de bien renverser la quille. Nous avons également prévu d'utiliser des gobelets dont la hauteur est inférieure à la hauteur h des rayons infrarouges du robot. Une fois renversée, une quille ne peut donc plus être détectée de nouveau (voir figure 9.2.a et 9.2.c).

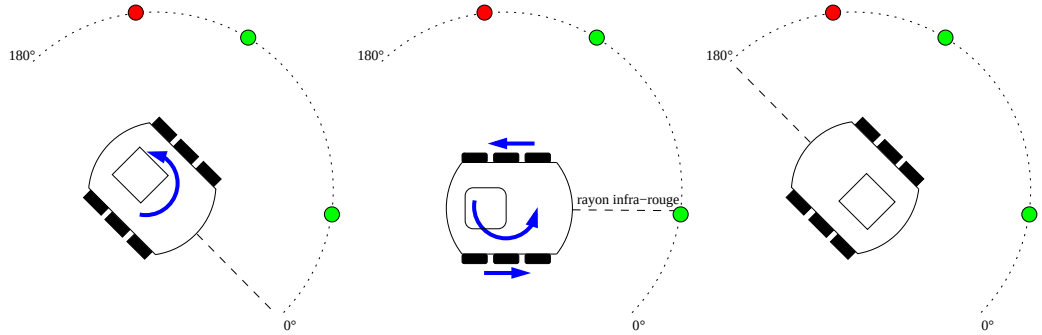
9.2 Le scénario

Notre première idée était de montrer qu'il est possible de gérer la consommation d'une ressource, en l'occurrence le temps, pour un scénario en robotique mobile avec le raisonnement progressif. Nous proposons un scénario original dans lequel les tâches sont modélisables sous forme de PRUs. Le robot devra détecter puis renverser une série de quilles disposées sur quatre

sites différents. Les deux modules que nous utilisons dans nos PRUs sont « détecter » et « renverser » une quille.

9.2.1 Description de chaque site

Chaque site sera représenté par un centre, sur lequel le robot se place avant de commencer à détecter les quilles, qui sont disposées sur un demi-cercle de rayon inférieur à la portée des capteurs infrarouges. Avant chaque expérience, les quilles sont placées à la main sur ce demi-cercle. Ceci introduit un degré d'incertitude quant au temps nécessaire pour détecter une quille. Il a donc fallu déterminer une fonction de distribution de probabilité de consommation de temps pour détecter une quille. Nous expliquerons comment nous avons procédé dans la section suivante. Pendant la mission, une fois arrivé sur un site, le robot se place en son centre, puis tourne sur lui-même de façon à s'orienter à un angle de 0 degré. Ensuite, il tourne sur lui-même vers la gauche et balaye un angle inférieur à 180 degrés. Dès que le capteur infrarouge détecte une quille, le robot peut décider de renverser la quille. Pour cela, il s'avance, renverse la quille, puis recule pour revenir exactement à la même place. Dès qu'il est de retour, ou s'il a décidé de ne pas bouger, il continue à tourner sur lui-même jusqu'à ce qu'il ait parcouru les 180 degrés du demi-cercle.



a) le robot se place **b)** il détecte une quille **c)** il a terminé

FIG. 9.3 – Les actions du robot sur un site.

9.2.2 Répartition des sites

Avant de commencer la mission, le robot connaît la répartition des quilles, et le nombre de sites. Dans notre cas, il y aura exactement quatre sites, sur lesquels seront disposées trois quilles, une verte, une seconde verte, puis une rouge, toujours dans cet ordre (voir figure 9.4).

Explications *pour la figure 9.4 : nous avons disposé sur quatre sites différents quatre séries identiques de trois quilles : 2 vertes puis une rouge. Le robot, initialement placé au centre de la salle a pour mission de gagner le plus possible de points en faisant tomber des quilles dans un*

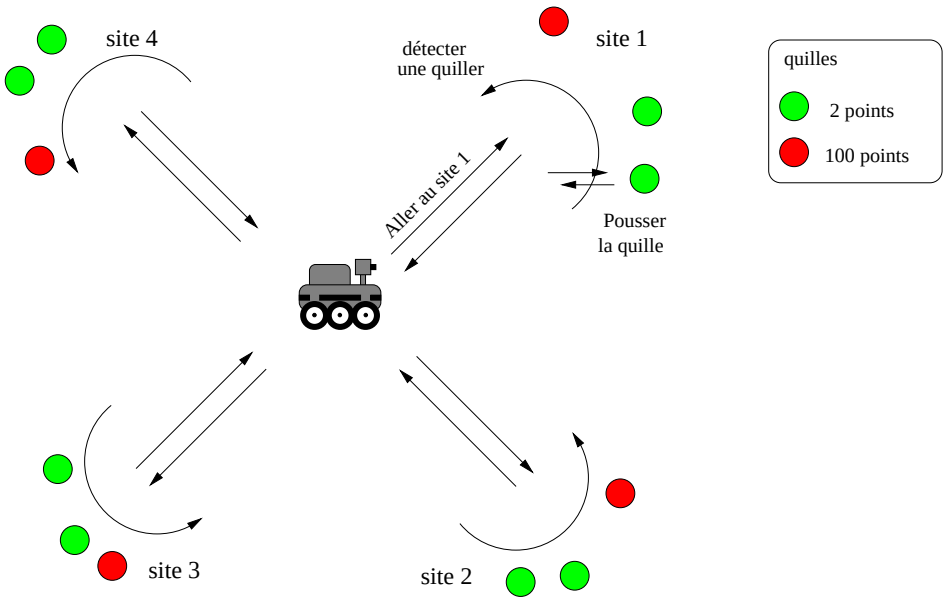


FIG. 9.4 – Le scénario de la mission bowling.

temps imparti, sachant que s'il renverse une quille verte, il gagnera 2 *points, et que s'il renverse une quille rouge il en marquera* 100.

9.2.3 But de la mission

Le but pour le robot est de marquer un maximum de points avant une date limite ou il devra s'arrêter. Pour cela, il devra renverser des quilles. Une quille verte renversée rapporte 2 points, une quille rouge 100. Détecter une quille ne rapporte pas de point. L'ensemble des actions à effectuer sur un site sont décrites dans une unité de raisonnement progressif.

9.3 Une Unité de Raisonnement Progressif pour un site

Afin que le robot sache dans quel ordre il doit effectuer les actions qui lui sont proposées nous avons créé une PRU qui décrit comment détecter puis renverser chacune des quilles du site. Les PRUs sont construites à partir des deux modules dont nous disposons « détecter » et « pousser ». Un troisième module, « ignorer », permet au robot de ne pas bouger.

Explications *pour la figure 9.5 : sur cette figure nous représentons la PRU à partir de laquelle notre robot calculera la politique qui lui permettra de décider quelles quille renverser pour obtenir un score maximal à la fin de la mission. La qualité de chaque module est inscrite dans un carré à droite de celui-ci. Pousser la quille au niveau* 2 *permet d'accumuler* 2 *points. Chaque module consomme plus ou moins de temps : détecter prend de* 1 *à* 10 *secondes, pousser de* 8 *à* 10 *secondes*

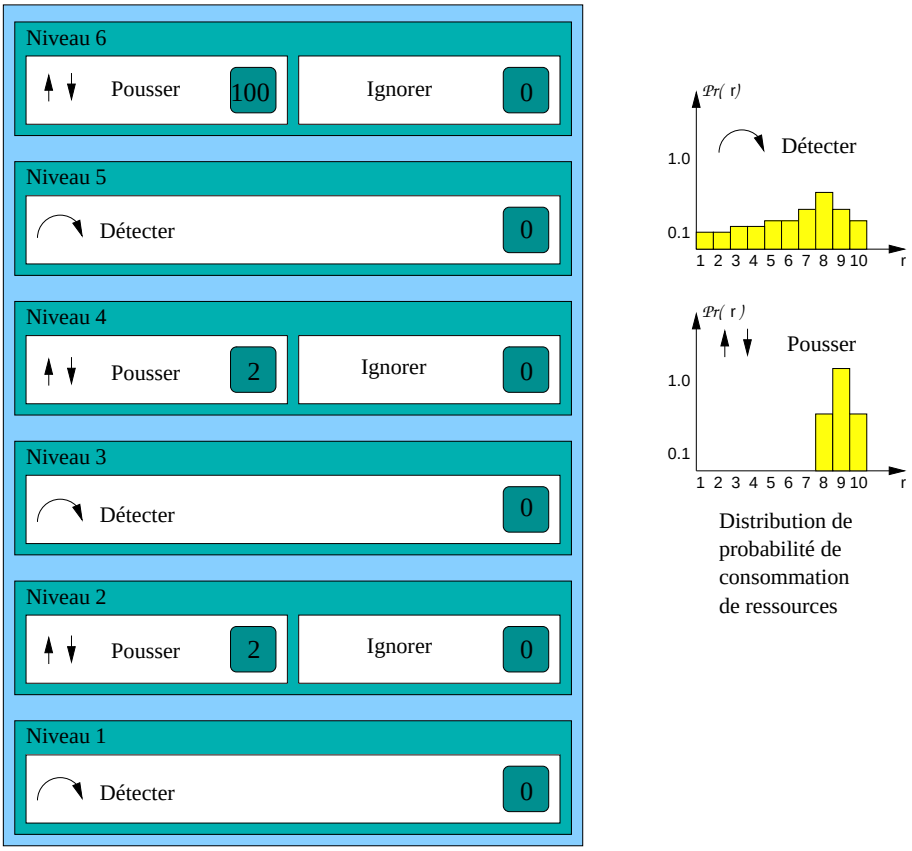


FIG. 9.5 – PRU pour la mission quille.

et ignorer ne consomme pas de temps (partie droite de la figure 9.5). La mission complète est composée de 4 PRUs de ce type.

9.3.1 Organisation des modules

Les PRUs sont là pour décrire au robot dans quel ordre il doit effectuer les différents modules. A chacun d'eux correspond une qualité précise (0, 2 ou 100). Chaque module consomme du temps. Cette consommation de temps est sujette à l'incertitude, nous expliquons dans le paragraphe suivant comment nous avons construit ces fonctions.

9.3.2 Les fonctions de distribution de probabilités de consommation de ressources

Comme les quilles sont placées à la main sur chaque site, le robot ne sait pas exactement combien de secondes il va lui falloir pour trouver la première quille. Cependant, nous sommes soumis à certaines contraintes pour l'expérience :

– l'arc de cercle mesure 180 degrés,

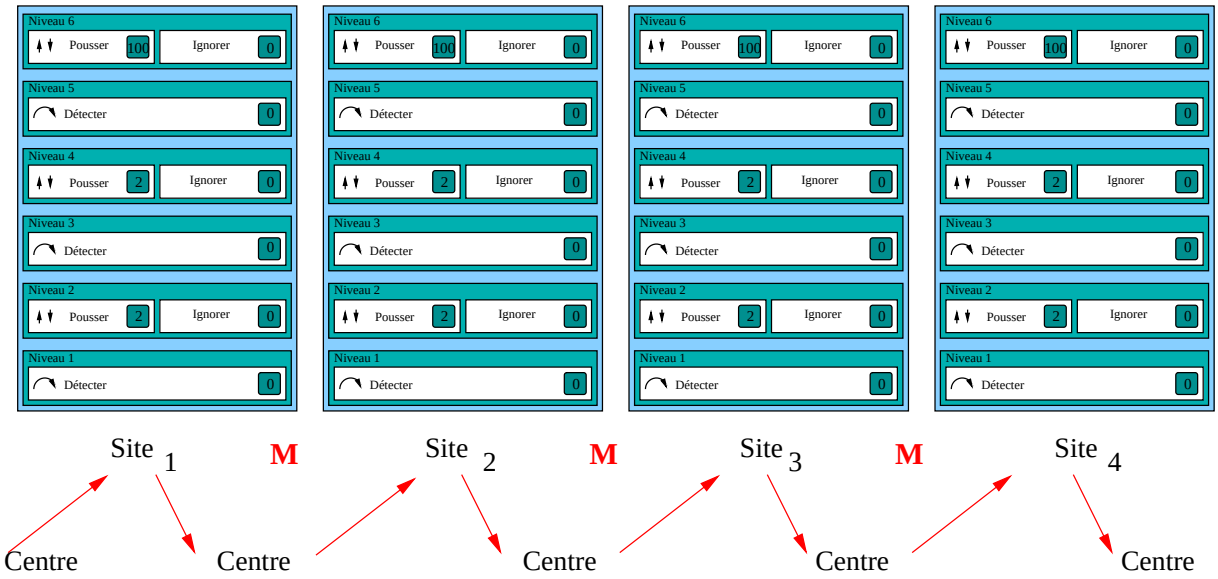


FIG. 9.6 – La mission quille formalisée avec le raisonnement progressif.

– les quilles doivent être distantes de plusieurs centimètres,

Comme l'arc de cercle mesure 180 degrés et que la vitesse de rotation du robot Koala est constante et fiable, nous savons en combien de temps le robot peut détecter les trois quilles. Les quilles doivent être distantes de plusieurs centimètres pour deux raisons : il faut que le robot puisse les distinguer, mais il faut également que lorsqu'il en renverse une, il ne renverse pas sa voisine. Il faut donc les espacer d'au moins 30 centimètres (la largeur du robot Koala). Nous avons donc divisé le temps de rotation totale du robot moins quelques secondes (car rares sont les cas où la dernière quille se trouve à 180 degrés) par trois (le nombre de quilles) pour obtenir le temps le plus probable pour détecter une quille (environ 8 secondes sur notre figure 9.5).

Il aurait été surement beaucoup plus rigoureux de faire un apprentissage en effectuant plusieurs séries de détections à la suite pour déterminer cette distribution de probabilité. Quoiqu'il en soit, la fonction que nous proposons semble manifestement bien simuler l'incertitude introduite par l'humain qui place les quilles.

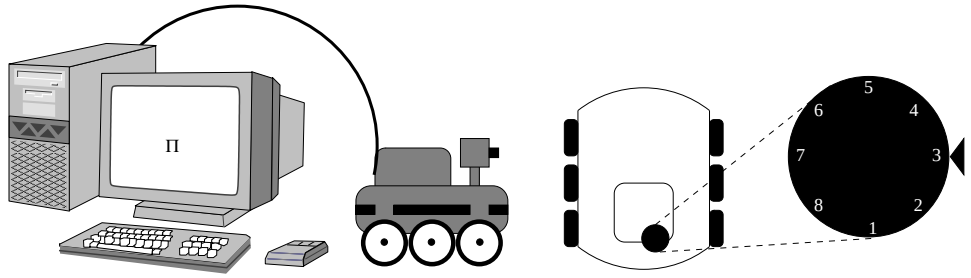
Pour le module « pousser », l'incertitude était plus simple à mesurer puisque la durée de cette action ne dépend que du robot Koala, dont le comportement est fiable. Le robot met toujours 9 secondes pour renverser une quille, à une seconde près.

9.3.3 La mission

La mission se divise en 4 sites, elle est donc composée de 4 PRUs identiques à celle de la figure 9.5.

Le déplacement entre les différents sites ne sera pas pris en compte lors de l'exécution de la mission, la prise de décision et la gestion de la ressource temporelle ne se fait que sur les sites. Le raisonnement progressif, tel qu'il a été présenté, ne permet pas de prendre en compte les déplacements. Nous en rediscuterons en fin de chapitre. Un compteur interne au robot mesure le temps qu'il lui reste pour exécuter la mission. Le démarrage d'une mission s'effectue en deux temps (voir figure 9.7) :

1. la politique est calculée sur un ordinateur puis transférée au robot,
2. nous fixons le temps imparti pour la mission grâce au curseur sur la rondelle noire.



a) calcul de la politique **b)** réglage du compteur temps

FIG. 9.7 – Démarrer une mission.

Nous laissons au robot un temps égal à 20 fois ce qui est indiqué sur le curseur pour effectuer la mission (sur notre figure il dispose donc de 60 secondes). Au bout de ce laps de temps, il s'arrête net. La politique est calculée une fois seulement sur l'ordinateur. Elle indique au robot ce qu'il a de mieux à faire en fonction de son état (temps restant, PRU courante, niveau, qualité accumulée). Nous pouvons faire plusieurs expériences en modifiant le curseur avec la même politique. Une fois le curseur ajusté, nous allumons le robot. La mission commence...

9.4 Un exemple d'exécution

Nous avons filmé le robot en action lors d'une mission bowling. Cette vidéo est disponible sur notre site internet à l'adresse suivante : `http://users.info.unicaen.fr/~slegloan/videos/robotbowling.AVI`. Nous vous présentons une capture de cette vidéo sur la figure 9.8. Nous lui avons alloué 80 secondes pour effectuer la mission (rappelons que le temps pour se déplacer de site en site n'est pas décompté). Nous avons répété cette expérience plusieurs fois.

Explications *pour la figure 9.8 : nous avons capturé plusieurs images de la vidéo présentant le robot qui effectue la mission : renverser des quilles. Sur la figure 9.8.a, le robot arrive sur le site* 2. *Nous voyons le robot prêt à détecter les quilles sur la figure 9.8.b. Sur la troisième photo, figure 9.8.c, le robot a détecté la troisième quille, qui est rouge et qui vaut* 100 *points. La dernière*

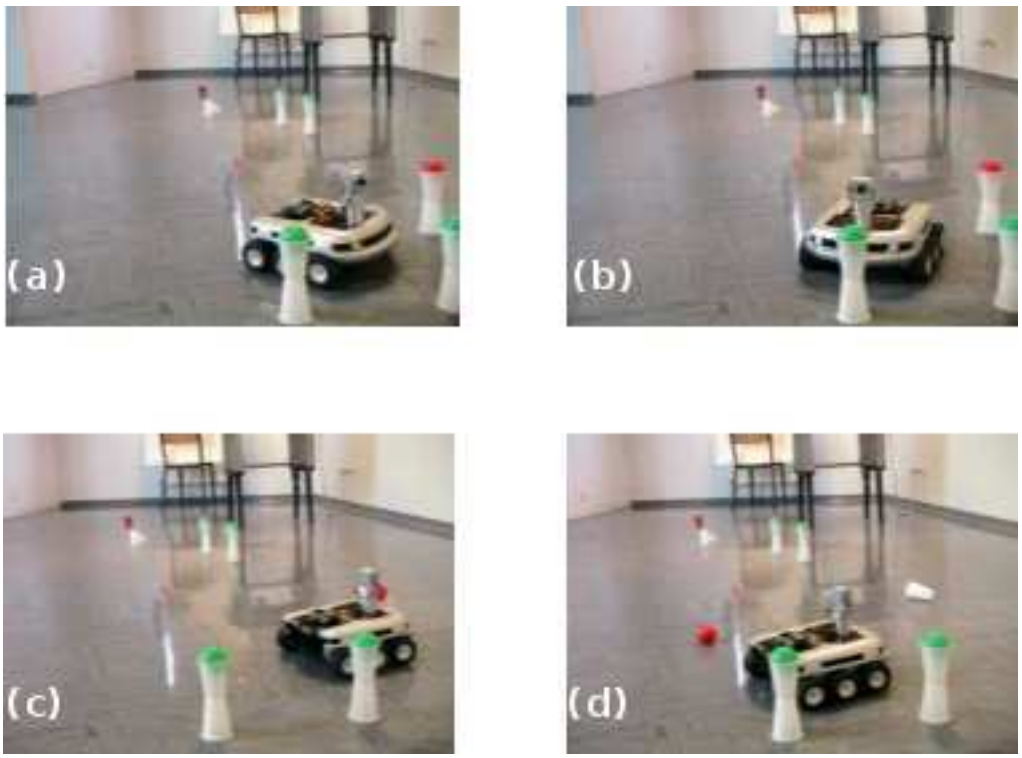


FIG. 9.8 – Le robot en action.

photo, figure 9.8.d, montre la quille rouge abattue. Le robot va ensuite se retourner, puis revenir au centre. Finalement, il ira pousser les quilles du site 3*, reviendra au centre, puis se dirigera vers le site* 4*, avant de finir la mission en revenant une dernière fois au centre. Ceci a déjà été illustré sur la figure 9.4.*

Le robot a toujours renversé au moins les quatre quilles rouges en 80 secondes, totalisant ainsi un score minimum de 400 points, qui était le résultat attendu compte tenu des distributions de probabilités de consommation de ressources et des qualités des modules de la figure 9.5. Dans la vidéo que nous avons faite, le robot a réussi à faire un score de 402 points. Il a réussi à économiser un peu de temps sur chacun des premiers sites, et finalement, il a renversé une quille verte en plus ! Cette économie a été réalisée car nous avions surement posé les quilles plus près les unes des autres que pendant les autres essais. Notre robot a donc réussi à s'adapter à la situation.

9.5 Discussion

Les résultats sont concluants, le robot semble avoir un comportement « intelligent ». Cependant, il faut admettre que notre expérience n'est pas une reproduction conforme de l'expérience du robot martien, mais sur la partie planification de tâches, notre résultat est probant. Un ro-

bot martien possède d'autres effecteurs et capteurs qui lui permettent d'évoluer sur un terrain accidenté. Notre préoccupation principale est de gérer les ressources consommables et planifier les tâches. Il n'est pas possible de prendre en compte les déplacements physiques du robot avec le raisonnement progressif. En effet, l'exécution des PRUs est interruptible, et une interruption permet de passer immédiatement à la PRU suivante. Si l'agent se déplace d'un point A à un point B et qu'il interrompt son mouvement en un point C, il faut qu'il revienne en A pour faire comme si rien ne s'était passé (voir figure 9.9). Il consommera donc du temps pour revenir, et l'état avant et après l'interruption seront différents. Dans le raisonnement progressif, après une action **M**, le montant de ressources ne change pas. L'aspect planification de chemin n'est pas pris en compte dans la version actuelle.

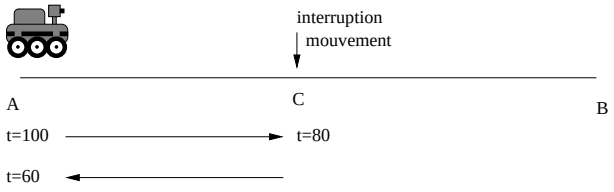


FIG. 9.9 – Interrompre un déplacement.

Le robot part du point A, il lui reste 100 secondes pour la mission. 20 secondes plus tard, il décide d'interrompre son mouvement. Pour revenir en A, il lui faut de nouveau 20 secondes, donc l'interruption a modifié le nombre de ressources restantes.

L'objectif que nous étions fixé est de gérer la consommation de ressources dans un environnement incertain. Nous pensons que les déplacements doivent être gérés par un planificateur de plus haut niveau. Le raisonnement progressif doit uniquement permettre à un agent d'adapter sa consommation de ressources au problème. Un objectif à long terme serait d'intégrer le raisonnement progressif dans un planificateur.

Nous aurions également voulu intégrer plusieurs ressources pour cette démonstration. Ceci n'a pas été techniquement faisable. Finalement, l'ajout et la suppression de tâches n'a pas été implémentée pour plusieurs raisons :

- le robot ne peut pas détecter de nouveau site puisqu'il n'a pas de caméra,
- le système embarqué du robot ne possède pas de système de fichiers.

Nous aurions pu simuler l'arrivée de nouvelles tâches en reliant le robot à un ordinateur portable (avec un fil), en lui indiquant chaque fois où et quand ces PRUs apparaissent. La nouvelle politique aurait également été calculée sur le PC. Un tel protocole expérimental ressemble plus à nos yeux à une retranscription directe du comportement d'un simulateur sur le robot qu'à une véritable expérience robotique. Certes, le robot se serait adapté à ce que l'ordinateur lui aurait ordonné, mais pas vraiment à l'environnement.

Conclusion

Nous avons construit un scénario pour la robotique mobile à partir des moyens disponibles dans le laboratoire, à savoir le robot Koala, et intégré un mécanisme de contrôle basé sur le raisonnement progressif et les processus décisionnels de Markov. L'expérience du robot Bowling nous montre bien que cette intégration est possible, que les tâches sont simples à modéliser, et que le robot est capable de s'adapter aux ressources qu'il lui reste.

Nous envisageons de poursuivre nos expériences en intégrant le mécanisme de recomposition dynamique de la politique, mais comme nous n'avons pour l'instant pas les moyens de détecter une nouvelle tâche (c'est-à-dire l'apparition d'un site supplémentaire), et que de plus le système informatique du robot Koala est limité (pas de système de fichier, pas de possibilité de communication par WiFi, pas de caméra...), nous projetons de mettre en place ces expériences plus tard.

Chapitre 10

Expériences menées dans le cadre du raisonnement progressif avec plusieurs ressources consommables

Introduction

Nous avons montré dans le chapitre 6 comment étendre le modèle du raisonnement progressif à plusieurs ressources. Dans le chapitre 5 nous avons montré qu'il est possible de ne pas stocker la politique pour toute la mission dans le robot, puisqu'il est possible de calculer celle-ci rapidement si l'on dispose de la fonction de valeur du premier niveau de la PRU qui suit dans la mission. Le problème est donc pour nous de calculer la fonction de valeur. Le problème majeur du calcul de la fonction de valeur pour le raisonnement progressif est la taille de l'espace d'états : l'algorithme est proportionnel à la taille de l'espace d'états qui devient grand lorsque le nombre de PRUs augmente dans la mission.

L'ajout de ressources supplémentaires aggrave le problème : l'espace d'états gagne une (voire plusieurs) dimension et le calcul de la fonction de valeur demande encore plus de ressources machine. Nous avons utilisé quelques propriétés de la fonction de valeur et la notion de facteur de ressource limitante pour que l'on puisse d'un côté calculer plus rapidement la fonction de valeur et de l'autre compresser sa représentation par une technique d'agrégation d'états par valeur.

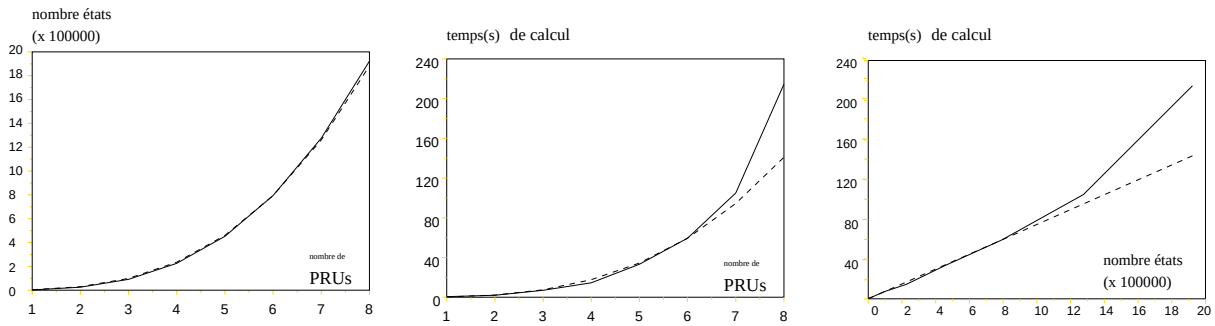
Puisque le robot Koala que nous avons présenté dans le chapitre précédent ne permet pas de faire d'expérience avec plusieurs ressources consommables, nous n'avons pas fait de tests expérimentaux grandeur nature. Nous voulons valider les algorithmes présentés dans le chapitre 6. L'objectif principal de ce chapitre est de montrer que notre algorithme de calcul de fonction de

valeur est beaucoup plus rapide que les algorithmes « classiques[49] ». Notre algorithme retourne la fonction de valeur optimale. Il n'est donc pas utile de comparer les politiques obtenues via les différents algorithmes, puisqu'elles sont identiques et optimales. Dans un dernier temps, nous présenterons les résultats de l'algorithme d'agrégation par valeur qui compresse l'espace d'états de la fonction de valeur.

10.1 Analyse de performance temporelle de l'algorithme de calcul de la fonction de valeur

L'introduction de nouvelles ressources nous oblige à faire intervenir plusieurs dimensions supplémentaires dans l'espace d'états. Puisque nous ne faisons pas d'expériences sur un robot, la nature de ces ressources importe peu. Il peut s'agir du temps, de l'énergie, de la mémoire, d'une quantité d'eau dans une citerne par exemple. Dès que l'on parlera de temps dans ce chapitre, ce sera du temps nécéssaire pour calculer la fonction de valeur et non pas une ressource consommable utilisée par le robot.

Nous avons calculé les fonctions de valeur optimales pour des missions où nous faisons intervenir deux ressources avec l'algorithme 7 : l'algorithme exhaustif par niveaux. Le temps de calcul devient particulièrement important (supérieur à trois minutes) dès lors que nous avons une mission composée de 8 PRUs (voir figure 10.1.**a**). Nous atteignons les deux millions d'états pour cette mission.



a) augmentation du nombre d'états en fonction du nombre de PRUs

b) temps de calcul nécéssaire en fonction du nombre de PRUs

c) augmentation du temps de calcul en fonction du nombre d'états générés

FIG. 10.1 – Analyse temps espace d'états pour des missions avec deux ressources.

Explications *pour la figure 10.1 : nous avons calculé les fonctions de valeurs qui permettent d'obtenir les politiques optimales locales pour des missions dont les longueurs varient entre* 1 *et* 8 *PRUs. La figure (**a**) indique la taille de l'espace généré par le calcul de la fonction de valeur en fonction du nombre de PRUs. La figure (**b**) indique le temps nécessaire pour calculer la fonction de*

[49] Algorithme 5 et algorithme 7.

*valeur en fonction du nombre de PRUs. La figure (**c**) illustre la relation linéaire entre le nombre d'états et le temps nécessaire pour calculer la fonction de valeur.*

Ces expériences ont été effectuées avec deux ressources, donc le nombre d'états augmente au cube avec le nombre de PRUs. Sur la figure 10.1.**a**, la fonction $k.(\#\text{PRUs})^3$, avec $k = 3\,700$ (en pointillés) approche le nombre d'états générés en fonction du nombre de PRUs (en trait plein). C'est ce que nous attendions dans le chapitre 6. Nous avons vu dans le chapitre 5 que la complexité du calcul de V est en $\mathcal{O}(\mathcal{S})$. Le temps de calcul devrait « théoriquemnt » être proportionnel au nombre d'états générés. Mais sur la figure 10.1.**c** nous voyons que cette proportion n'est respectée que jusqu'à 6 PRUs. Au-delà le calcul passe en mémoire virtuelle faussant la mesure de comparaison temporelle. Le temps de calcul (figure 10.1.**b**) augmente lui aussi avec le nombre de PRUs au cube jusqu'à 6 PRUs, et un peu plus au-delà. Nous n'avons pas inscrit sur ces courbes les résultats obtenus pour une mission composée de plus de 8 PRUs parce que nous atteignons ici les limites de notre machine.

Nos calculs ont été effectués sur une machine dont les caractéristiques sont les suivantes :

- Intel(R) Pentium(R) 4 CPU 2.60GHz
- 512 MB de mémoire vive

Le langage de programmation utilisé est Python.

Dans ce qui va suivre, nous allons montrer que notre algorithme peut calculer la même fonction de valeur beaucoup plus rapidement.

10.2 Algorithme accéléré pour le calcul de la fonction de valeur

La complexité de l'algorithme de calcul de fonction de valeur pour une mission composée de PRUs est proportionnelle à l'espace des états possibles pour cette mission. Nous avons présenté dans le chapitre 5 deux algorithmes de programmation dynamique dont la complexité est similaire pour calculer cette fonction de valeur : par exploration de l'espace d'états (chainage avant puis arrière) ou par niveau (avec un chainage arrière). Dans le chapitre 6, nous avons implémenté plusieurs algorithmes basés sur l'algorithme par niveau :

- un algorithme accéléré qui ne calcule pas les *Bellman Backup* pour les états appartenant à un palier et qui génère tous ces états en leur affectant directement la bonne valeur,
- un algorithme qui agrège pendant le calcul de la fonction de valeur chaque état d'un palier dans un seul et même état et qui évite aussi par la même occasion de faire le *Bellman Backup*,
- un algorithme où chaque sous-espace d'états est agrégé de façon à ce qu'il ne reste qu'un seul état agrégé par valeur possible différente dans la fontion de valeur sur ce sous-espace.

Nous présentons les performances temporelles de l'algorithme par programmation dynamique basique, l'algorithme accéléré sans agrégation et l'algorithme avec agrégation faible, pour des

missions dont la longueur varie entre 1 et 10 PRUs de type 1 sur la figure 10.2. Nous ne présentons pas les performances temporelles de l'algorithme avec une agrégation forte, puisque les résultats sont obtenus dans un temps largement supérieur à l'algorithme basique (ceci est dû, rappelons-le, à la complexité nécessaire pour retrouver une information dans un sous-espace d'états lors du calcul de la valeur de chaque état).

Nous obtenons des résultats similaires pour les PRUs de type différents c'est-à-dire un temps de calcul réduit d'un facteur 3 environ pour l'accélération sans agrégation et par 10 avec une agrégation faible.

L'allure des fonctions de valeur obtenues pour des sous-espaces d'états où le niveau et la qualité sont fixés nous permet de voir qu'un tiers des états environ appartiennent à un palier relatif à $\mathtt{r}_1$, qu'un autre tiers d'états appartient à un palier relatif à $\mathtt{r}_2$ et que le reste des états constitue un tiers dont la valeur doit être calculée. Ceci explique le facteur de réduction par 3 quand on évite le *Bellman Backup* sur les paliers. Comme l'espace mémoire est moins encombré avec l'agrégation faible, nous obtenons des résultats encore plus rapidement.

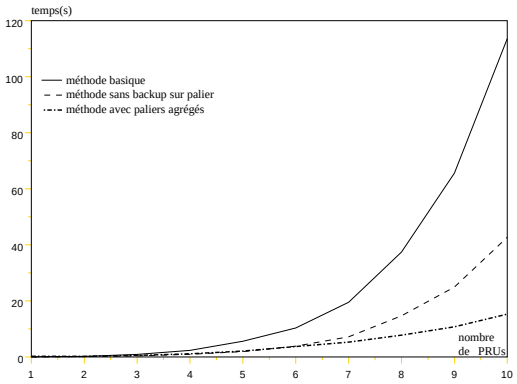


FIG. 10.2 – Temps de calcul de la fonction de valeur optimale avec deux ressources pour n PRUs.

Explications *pour la figure 10.2 : cette figure représente le temps nécessaire pour des missions dont la longueur varie entre* 1 *et* 10 *PRUs de type* 1. *L'algorithme avec agrégation faible ne prend que 10 secondes pour 10 PRU alors qu'il fallait pratiquement deux minutes avec un algorithme de programmation dynamique basique.*

En utilisant l'algorithme accéléré avec agrégation faible, nous pouvons calculer des fonctions de valeur pour des missions composées de 20 PRUs (de type 1) en une minute trente secondes. La méthode classique demande une minute quarante secondes pour calculer la fonction de valeur d'une mission composée de 10 PRUs seulement. Nous prétendons donc pouvoir calculer des fonctions de valeurs pour des missions deux fois plus longues qu'avant.

Rappel : Les fonctions de valeur obtenues par les trois méthodes sont les mêmes et optimales. Nous avons bien vérifié leur égalité une fois leur calcul effectué.

Nous avons également représenté le nombre d'états générés par la méthode de programmation dynamique basique et la méthode par agrégation faible sur la figure 10.3. La méthode par agrégation faible génère 30000 états pour calculer la fonction de valeur d'une mission composée de 10 PRUs alors que la méthode classique en génère presqu'un million. L'espace mémoire occupé pendant le calcul de la fonction de valeur est nettement inférieur lorsqu'on utilise la méthode d'agrégation faible.

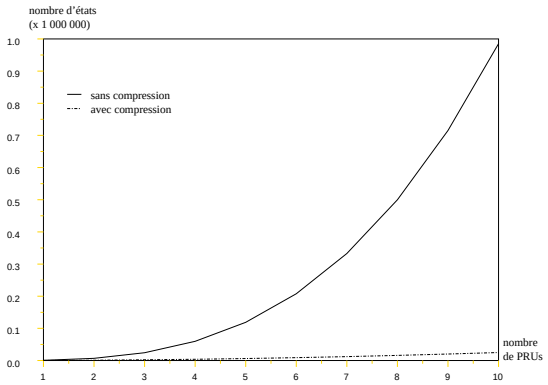


FIG. 10.3 – Nombre d'états générés pour le calcul de la fonction de valeur optimale avec deux ressources pour n PRUs.

10.3 Représentation structurée de la fonction de valeur pour les ressources multiples

Nous avons proposé dans le chapitre 6 une structure permettant de représenter la fonction de valeur associée à un espace d'états (à un niveau `n` et à **Q** fixés) de façon plus compacte. L'algorithme avec agrégation forte n'étant pas utilisable pendant le calcul de la fonction de valeur[50], nous le présentons comme un algorithme de représentation compressé de la fonction de valeur. Cet apport est un petit plus qui permet d'économiser de l'espace mémoire sur le robot, une fois cette fonction calculée.

Cette représentation ne génère aucune perte de données, c'est une agrégation exacte et non uniforme. Nous pouvons donc retrouver la valeur exacte de n'importe quel état après avoir changé de représentation.

L'idée est de ne pas garder en mémoire la valeur de tous les états du niveau à représenter mais plutôt de représenter tous les états qui ont la même valeur par les bornes inférieures (au sens de la relation de dominance partielle) de cet espace. L'algorithme de compression est présenté dans le chapitre 6. Nous donnons sur la figure 10.4 une représentation tridimensionnelle de la fonction

[50] À cause de la complexité du temps d'accès à une donné dans un sous-espace agrégé de cette façon, cf chapitre 6.

de valeur pour les états avant le premier niveau de la première PRU d'une mission composée de 5 PRUs. En d'autres termes, cette fonction représente la fonction de valeur pour le sous-espace des états de départ possibles pour cette mission.

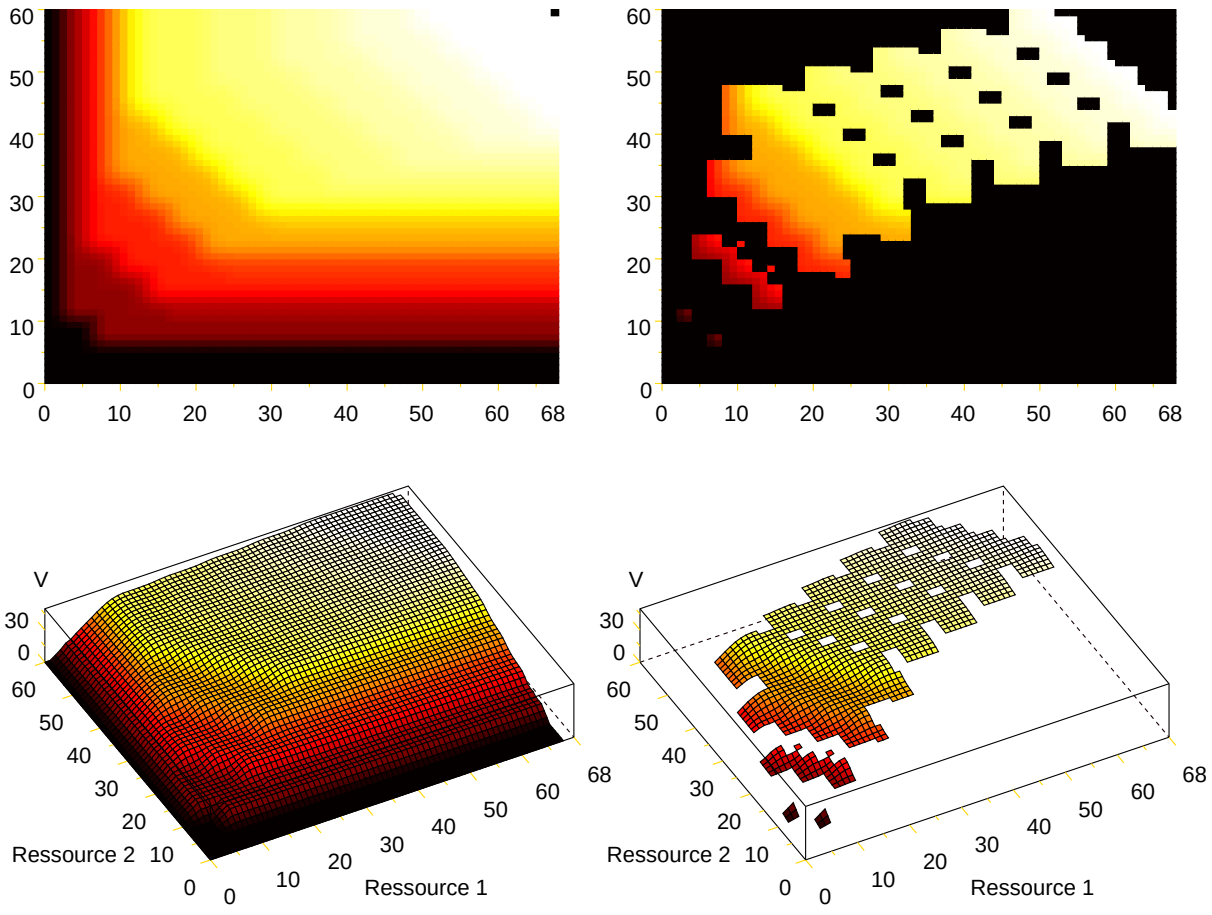


FIG. 10.4 – Compression de l'espace d'états pour les ressources multiples.

Explications *pour la figure 10.4 : Après avoir calculé la fonction de valeur optimale pour une liste de 4 PRUs, nous avons tracé la fonction de valeur sur le sous-espace des états de départ possibles pour cette mission ($\mathcal{S}_{0,0,0}$). Les valeurs les plus basses correspondent aux couleurs les plus sombres. Á droite de cette figure, nous avons représenté les états qu'il faut conserver pour réussir à reconstituer la courbe de gauche. On voit que les paliers dûs à la ressource limitante ont disparu, les paliers dûs aux modules dont la consommation de ressources est éloignée quant à eux font des "trous" dans la courbe. Tous les états représentés à droite sont stockés et suffisent à retrouver la vraie fonction de valeur. Ce processus est identique sur tous les sous-espaces d'états de la mission, à* $\mathtt{p}, \mathtt{n}, \mathbf{Q}$ *fixés.*

Les états qui doivent être stockés pour retrouver la valeur de n'importe quel autre état, c'est-à-dire les états qui bornent les régions de valeur, sont représentés sur la partie droite de la

figure 10.4. Les états qui ne sont pas stockés dans la structure ne sont simplement pas représentés.

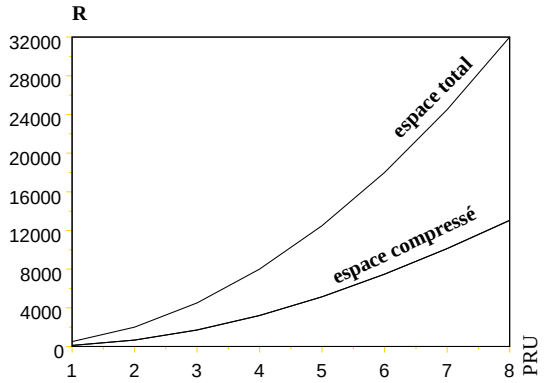


FIG. 10.5 – Compression du nombre d'états stockés en mémoire pour des missions de longueurs 1 à 8 PRUs.

Explications *pour la figure 10.5 : nous avons compressé la fonction de valeur sur leurs espaces d'états obtenus pour pour des missions dont les longueurs varient entre* 1 *et* 8 *PRUs. Le taux de compression est d'environ trois. On ne stocke pas les paliers dûs au facteur de ressource limitante, ni ceux dûs aux modules "éloignés".*

La figure 10.5 nous donne le nombre d'états qu'il faut stocker pour représenter la fonction de gain espéré, soit en conservant tous les états (espace total) soit en ne gardant pour une valeur que les états clés qui forment la borne inférieure de chaque région de valeur (espace compressé). Ces clés sont les états qui définissent les $\phi_v^{\min}$ pour chaque valeur v possible de la fonction. Le gain (en espace de stockage) apparaît nettement sur la courbe. Plus le nombre de PRUs est important, plus le nombre d'états augmente, mais aussi plus la compression d'espace d'états est intéressante.

Cette technique d'agrégation permet de gagner en espace mémoire mais son utilisation pour le calcul n'est pas possible. Elle servira donc juste de représentation compacte de la fonction de valeur sur un sous-espace d'états.

Conclusion

L'extension du raisonnement progressif s'est déroulée en trois parties. Dans le chapitre 6, nous avons étendu le **formalisme** en ajoutant plusieurs dimensions supplémentaires à la ressource unique qui devient un **n-uplet** de ressources. Dans ce même chapitre, nous avons proposé un **algorithme** qui permet d'**accélérer** le calcul de la fonction de valeur en profitant des propriétés de cette fonction et en agrégeant les états paliers qui bornent les sous-espaces d'états de la mission.

Avec cette même idée, nous avons mis au point une structure de **représentation agrégée**

plus forte de la fonction de valeur sur les sous-espaces d'états ; elle nous évite de stocker sur le robot la fonction de valeur entière. Un autre algorithme permet de recomposer la fonction de valeur exacte à partir de l'espace agrégé.

Ce chapitre valide les performances des algorithmes que nous venons de citer. Nous avons vu que l'algorithme de calcul de la fonction de valeur rapide avec agrégation faible permet de diminuer considérablement le temps de calcul ainsi que l'espace mémoire utilisé. La compression forte de l'espace d'états est un petit plus qui permet de stocker seulement un tiers des états possibles de la mission sur le robot. Ces états lui serviront à calculer sa politique au fur et à mesure qu'il exécutera les PRUs dans la mission. L'algorithme de recomposition de la fonction de valeur est suffisamment rapide pour permettre au robot de fonctionner de cette façon.

L'ajout de ressources supplémentaires rallonge inévitablement le temps de calcul nécessaire pour obtenir la fonction de valeur qui permet à notre robot de décider au bon moment quelle action effectuer mais, malgré cela, nous avons montré qu'il était possible de répondre partiellement au problème de l'explosion de l'espace d'états, en utilisant un sens de parcours ad-hoc pour nos algorithmes et en tirant parti des propriétés de la fonction de valeur des états possibles de la mission.

Chapitre 11

Performances de l'algorithme de recomposition dynamique

Introduction

Nous avons présenté dans le chapitre 7 le cadre de notre problème concernant la planification dynamique. L'idée est de mettre en place rapidement un mécanisme de contrôle sur notre robot autonome lorsque la séquence de tâches de la mission change, de façon à ce qu'il puisse adapter sa consommation de ressources et maximiser son critère de gain. Ce système se veut doublement adaptatif : la première adaptation concerne l'incertitude de consommation de ressources. La deuxième adaptation concerne les changement possibles dans la mission. Dans le chapitre 5 nous avons vu qu'il était possible de séparer le calcul de la politique en deux parties, la première partie concerne l'évaluation de la fonction de gain espéré pour les tâches qui viennent après la tâche courante dans la mission, l'autre partie permet de construire une politique pour la PRU courante à partir de cette fonction de gain espéré. Dans le chapitre 8 nous avons montré qu'il était possible d'évaluer rapidement une fonction de valeur pour les tâches qui se situent après la tâche courante dans la mission, en effectuant une approximation de celle-ci.

Dans ce chapitre, nous allons :

- évaluer les performances de l'algorithme permettant de calculer une fonction de gain espéré approchée,
- comparer les fonctions de gain espéré approchées $V^{\sim}_{methode}$ à la fonction optimale V^* obtenue par programmation dynamique classique,
- expliquer comment comparer les politiques locales obtenues,
- comparer les politiques locales obtenues par approximation à la politique optimale, par la comparaison de Q-valeurs.

11.1 Performance temporelle

Le calcul de gain espéré approché $V^{\sim}_{morceaux\,profils}$ est instantané en utilisant notre méthode. Dans le tableau suivant, nous avons inscrit les temps de calcul pour des missions dont la taille variait entre 20 et 140 PRUs. La PRU utilisée à titre d'exemple est celle de la figure 1 page 195. Les résultats sont valables pour n'importe quel type de PRU. Quelle que soit la longueur de la séquence de tâche, notre algorithme donne un résultat en un temps largement inférieur à celui nécessaire pour obtenir cette même fonction avec l'algorithme de programmation dynamique classique (figure 11.1). Il suffit de quelques millisecondes pour calculer $V^{\sim}_{morceaux\,profils}$.

Nombre de PRUs	20	40	60	80	100	120	140
Méthode classique (en s)	3	7	16	35	63	112	187
Méthode dynamique (en ms)	0,4	0,9	1,2	1,7	2,2	2,6	3,5

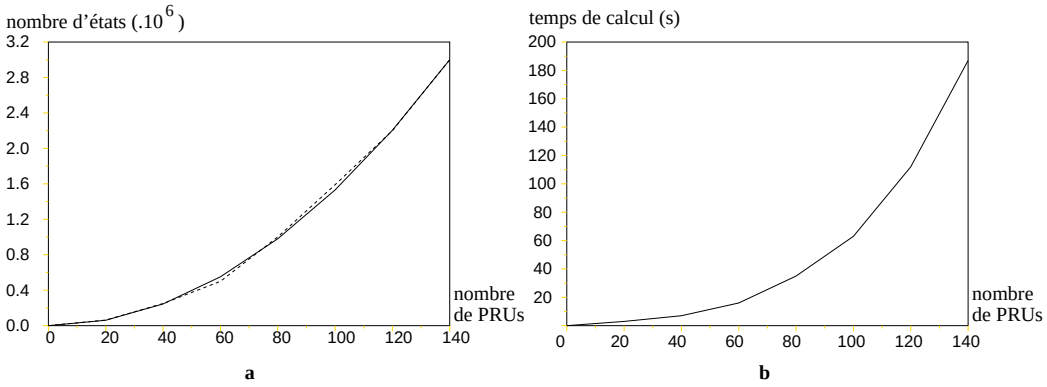


FIG. 11.1 – Complexité de l'algorithme classique.

Explications *: la figure 11.1.**a** représente le nombre d'états générés en utilisant un algorithme classique, et nous avons tracé en pointillés une parabole* $p : \mathbb{N} \to \mathbb{N}$, $x \to k.x^2$, $k = 154$. *Le nombre d'états varie bien au carré avec le nombre de PRUs dans la mission. La figure 11.1.**b** représente graphiquement le temps nécéssaire pour calculer la fonction de valeur optimale pour les missions du tableau. Nous ne représentons pas le temps mis par l'algorithme dynamique, puisque celui-ci est de l'ordre de la milliseconde.*

Cependant, cette approximation n'est pas la fonction V optimale. Nous comparons donc dans la section suivante les fonctions de gain espéré obtenues par approximation à l'optimal. Après celà, nous comparons les politiques calculées dans la PRU_0 en utilisant comme gain espéré une fonction approchée et la fonction optimale.

11.2 Comparaison des gains espérés $V^{\sim}$ et V^*

Puisque la politique locale obtenue dépend de la méthode utilisée pour obtenir le gain espéré pour le reste de la mission, nous avons commencé par comparer les fonctions de gain espéré obtenues par la méthode classique : $V^* : \mathtt{R} \rightarrow \mathbb{R}$ et par la méthode de recomposition rapide : $V^{\sim}_{methode} : \mathtt{R} \rightarrow \mathbb{R}$. Dans la section suivante, nous calculons puis comparons les politiques obtenues localement dans une PRU en tenant compte du gain espéré.

Après avoir calculé pour une séquence de PRUs la fonction de gain espéré optimal et approché, nous comparons les valeurs obtenues. Nous définissons une séquence par deux paramètres :

- Le type (ou les types) de PRUs présentes dans la séquence.
- La longueur de la séquence.

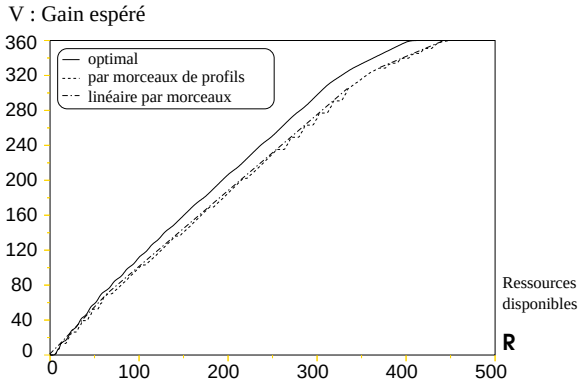


FIG. 11.2 – Comparaison de courbes de valeurs pour une mission de 20 PRUs de type différents.

Explications *pour la figure 11.2 : nous avons calculé et comparé la fonction de gain espéré d'une mission composée de 4 types de PRUs différentes obtenue avec l'algorithme de calcul de programmation dynamique classique, avec l'algorithme de recomposition par morceaux de profils de performance et finalement avec l'algorithme de recomposition linéaire par morceaux. Cette mission comporte 20 PRUs. La fonction V présentent beaucoup d'irrégularités (puisqu'on ajoute des profils et non des morceaux de droite), tandis que l'approximation linéaire par morceaux semble mieux suivre la fonction de gain espéré optimale V^*.*

Les deux fonctions, optimale et linéaire par morceaux, ont la même allure, sont plus ou moins parallèles, leur croissance est quasi équivalente. Ce résultat est très important, puisque lors du calcul de la politique locale, la valeur d'un état dépend de la différence entre deux points de la courbe de gain espéré de la suite de la mission. Ce qui est vraiment intéressant, ce n'est pas d'avoir de faibles écarts de valeurs entre les deux courbes, mais bien d'avoir de faibles écarts relativement aux variations de la valeur. La fonction obtenue par ajout de morceaux de profils de performance est trop irrégulière pour refléter la variation de la fonction de gain espéré optimale.

Nous avons également testé notre algorithme sur des missions de longueurs différentes : le résultat reste le même, les courbes de fonctions de gain espéré optimale et par approximation linéaire par morceau sont sensiblement parallèles.

Après avoir effectué différents tests de calcul de politique locale avec les deux fonctions approchées, nous avons effectivement constaté que la fonction linéaire par morceaux donnait de meilleurs résultats que la fonction par morceaux de profils de performance. Nous avons donc décidé de ne retenir que cette méthode d'approximation de gain espéré : l'approximation linéaire par morceaux.

11.3 Analyse des politiques locales obtenues

Afin de mesurer l'impact de notre fonction de gain espéré calculée dynamiquement sur la politique générée localement, nous allons comparer les politiques obtenues via les deux méthodes π^* et $\pi^\sim$ (voir étape 3 du chapitre 8).

Notez que nous ne pouvions pas comparer directement la valeur des états obtenue avec le calcul des différentes politiques, puisque le critère sur lequel se base le calcul de chaque politique n'est pas le même. Dans un cas, avec le gain espéré optimal V^*, nous avons un critère de maximisation de la récompense à long terme et, dans l'autre cas, notre fonction de gain espéré $V^\sim$ ne se base sur aucun critère précis. C'est pourquoi nous avons du faire appel à un système de comparaison basé sur les Q-valeurs de la politique optimale.

Attention : le processus de comparaison se base sur les Q-valeurs de LA politique optimale, on ne peut pas non plus comparer les Q-valeurs obtenues par la politique approchée à celle de la politique optimale, pour les raisons que nous venons d'évoquer plus haut. La comparaison va fonctionner en suivant les étapes :

1. calcul des gains espérés V^* et $V^\sim_{methode}$ pour une même mission,
2. calcul des politiques π^* et $\pi^\sim_{methode}$ pour la même PRU_0 locale, en utilisant le gain espéré correspondant,
3. pendant le calcul de π^*, garder **toutes** les Q-valeurs associées à chaque couple $(\mathbf{s}, \mathbf{a})$,
4. pour chaque état possible $\mathbf{s}$ de la PRU_0 locale, mesure de la différence entre la Q-valeur optimale et la Q-valeur dans la politique optimale associée à l'action liée à cet état dans la politique $\pi^\sim_{methode}$.

Nous pouvons de cette manière obtenir la perte en valeur espérée causée par une décision prise par le robot lorsqu'il suit une politique $V^\sim_{methode}$ par rapport à la décision qu'il aurait prise s'il avait connu la politique optimale dans un état donné de la PRU_0 locale. La valeur de la Q-valeur associée à un choix relatif à une politique approchée est inférieure ou égale à la valeur de la Q-valeur optimale, puisque pour π^*, c'est toujours la Q-valeur maximale qui est retenue. En un

état donné, l'erreur de Q-valeur est la différence entre la Q-valeur de la politique optimale liée à l'action optimale et la Q-valeur de la politique optimale liée à l'action de la politique approchée.

$$\forall \mathbf{s}, Erreur_{\mathcal{Q}}(\mathbf{s}) \quad = \quad \mathcal{Q}(\mathbf{s}, \pi^*(\mathbf{s})) - \mathcal{Q}(\mathbf{s}, \pi^{\sim}_{methode}(\mathbf{s})) \tag{11.1}$$

Les courbes d'erreurs que nous présentons sont la différence des Q-valeurs sur l'ensemble des états du premier niveau de la PRU_0 locale (c'est-à-dire $\mathcal{S}_{\mathbf{0},0,0}$) entre une de nos méthodes et la Q-valeur optimale dans cet état. Sur ce sous-espace d'états, seul $\mathtt{r}$ varie, donc nous pouvons présenter nos résultats sous forme de courbes allant de $\mathtt{0}$ à $\mathtt{r}_{\max}^{mission}$.

Nous avons regroupé et interprété les résultats pour plusieurs types de missions et de PRUs locales.

11.3.1 Les différentes approximations possibles pour le gain espéré V

Nous avons repris toutes les méthodes du chapitre 8 pour calculer rapidement le gain espéré à savoir l'approximation linéaire simple $V^{\sim}_{lineaire}$, l'approximation par ajout de morceaux de profils de performance $V^{\sim}_{morceaux\,profils}$ et l'approximation par ajout de morceaux linéaires $V^{\sim}_{morceaux\,lineaire}$. Nous rajoutons une fonction nulle $V^{\sim}_{nulle}$ qui vaut zéro en tous points qui représente un futur où l'agent ne peut plus rien gagner. Nous allons en effet comparer nos politiques approximatives à la politique optimale et nous voulions aussi comparer nos erreurs à celle commise par une politique gloutonne $\pi^{\sim}_{nulle}$. La politique gloutonne $\pi^{\sim}_{nulle}$ consistera à prendre la décision qui permet d'obtenir une valeur maximale dans la PRU locale, sans rien économiser comme ressources, sans se soucier de ce qui pourra se passer dans le futur.

11.3.2 Aucune erreur possible

Pour certaines missions couplées avec une PRU locale d'un type donné, aucune erreur n'est possible. C'est la cas notamment lorsque le rendement pour le reste de la mission est inférieur au rendement de la PRU locale. Autrement dit, si le gain espéré futur divisé par les ressources nécessaires pour obtenir ce gain est inférieur à la valeur du gain local divisé par les ressources pour obtenir ce gain, le reste de la mission (V^*) est négligeable devant ce qui peut être gagné localement. L'agent aura en effet dans ce cas de figure intérêt à exécuter le maximum de modules possibles dans la PRU courante (en maximisant le critère de somme de qualité obtenues). Nous éviterons donc d'extraire des résultats de telles expériences, qui conduisent à une erreur nulle quelle que soit la méthode d'approximation adoptée.

11.3.3 Une mission composée de PRUs homogènes

Nous disons qu'une PRU est *homogène* si les modules qui la compose sont du même ordre de grandeur, c'est-à-dire si la qualité accumulée ne diffère pas trop selon les différents modules et si les distributions de probabilités de ressources sont normales. Pour une mission composée uniquement de PRUs homogènes identiques, nous obtenons (figure 11.3 en haut à gauche) un gain espéré approché très proche du gain espéré optimal. La politique calculée partir de $V^{\sim}_{morceaux\,lineaire}$ est pratiquement similaire à la politique optimale et le nombre d'erreurs (en terme de Q-valeurs) est très faible (figure 11.3 en bas à droite).

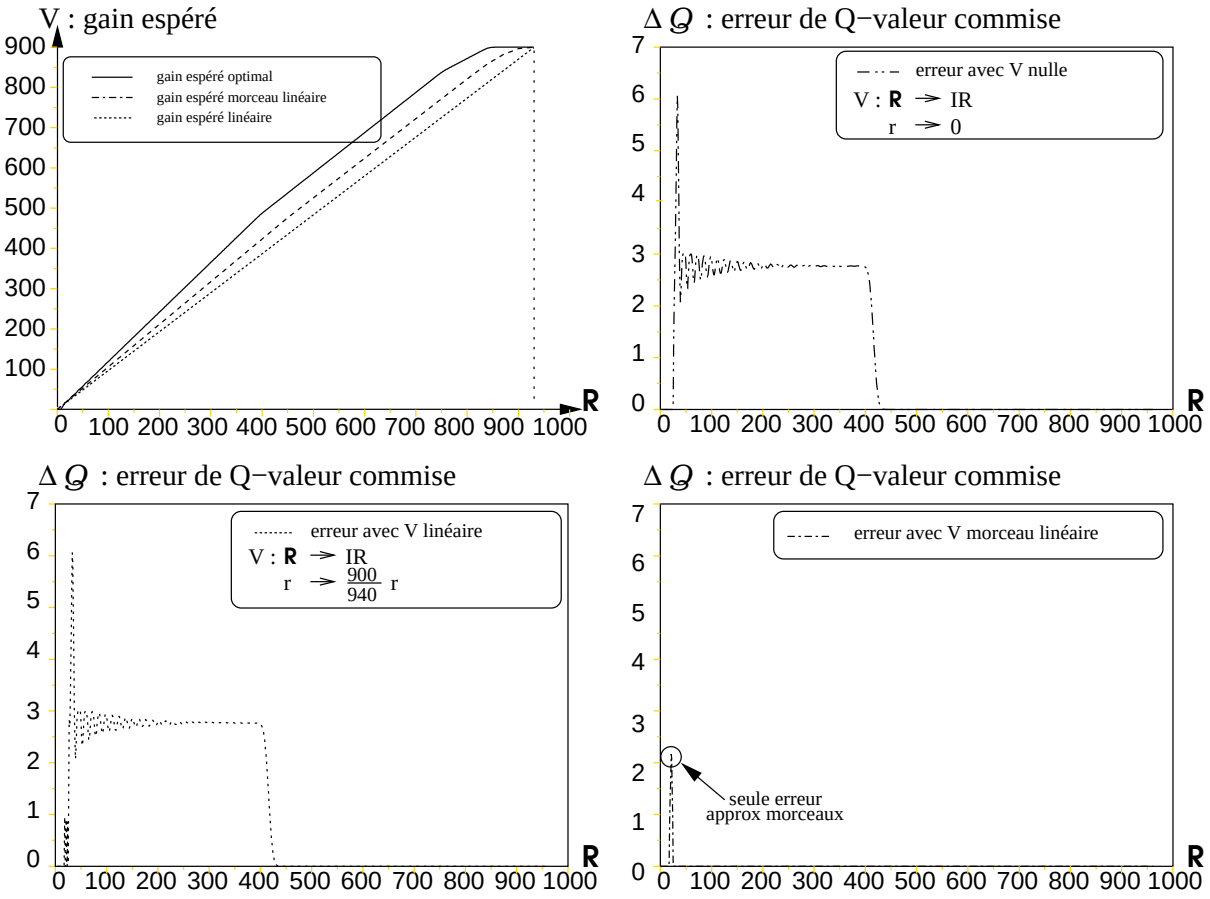
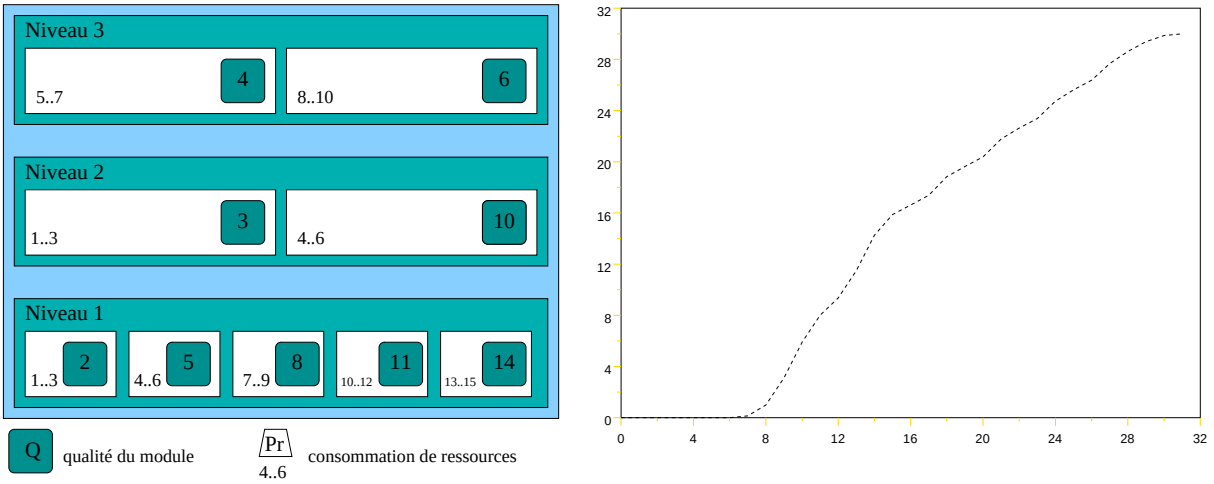


FIG. 11.3 – Une mission composées de nombreuses 30 PRU homogènes identiques.

Explications *pour la figure 11.3 : cette mission est composée d'une trentaine de PRUs de type* 1 *(toutes identiques) et la PRU locale est aussi de type* 1 *(voir figure 1). Sur la figure de gauche nous avons représenté le gain espéré pour le reste de la mission obtenu avec trois méthodes. La méthode de gain linéaire par morceaux s'approche plus du gain optimal que la méthode linéaire simple. En haut à droite, nous avons calculé la différence entre les Q-valeurs optimales maximales et les Q-valeurs optimales obtenues en appliquant les actions indiquées si la fonction de gain*

espéré est nulle. C'est une mesure de la perte en Q-valeur qui aura effectivement lieu si l'agent sait qu'il n'y a plus rien à gagner dans le futur. Si par exemple il reste 300 *unités de ressource à l'agent, il va prendre une décision qui lui fera perdre un gain de* 2.6 *s'il se base sur un gain espéré nul. En bas à gauche, les différences de Q-valeurs sont calculées en comparant une politique dont le calcul se base sur un gain espéré linéaire. Finalement, en bas à droite, il y a les erreurs en Q-valeur générées par notre méthode de recomposition linéaire par morceaux. Les deux méthodes nulle et linéaire pour obtenir une politique approchée génèrent de nombreuses erreurs pour les états pour lesquels il reste entre* 0 *et* 400 *unités de ressource. L'erreur générée par ces méthodes est autour de* 3 *et il y a un pic à* 6*. Notre méthode d'approximation linéaire par morceaux ne génère qu'une seule erreur sur le graphique en bas à droite, qui fait un pic à* $2,2$*.*



a) la PRU

b) profil de performance

FIG. 11.4 – Une PRU homogène.

Notre approche d'approximation linéaire par morceau du gain espéré ne génère qu'une seule erreur ! (indiquée sur la figure 11.3 en bas à droite). Dans ce cas, notre approche est très performante, meilleure que l'approximation linéaire simple $V^{\sim}_{lineaire}$. Cependant, ce type de mission n'est pas vraiment *parlante*, puisqu'un changement dans la mission ne peut faire varier que la longueur de celle-ci vu que toutes les PRUs sont du même type.

Modules proches

Nous avons fait la même expérience avec une mission composée de PRUs homogènes mais non identiques : l'écart-type de la qualité des modules qui les compose est faible et l'écart-type de la consommation de ressources sur l'ensemble de modules qui compose cette mission est aussi faible. Dans ce cas, les résultats sont très bons ; nous avons rassemblé les résultats sur la figure 11.5.

Nous ne voyons malheureusement[51] aucune erreur de Q-valeur pour la politique $\pi^{\sim}_{morceaux\,lineaire}$ sur le graphique en bas à droite. Nous avons fait varié le nombre de PRUs différentes, la taille de la mission, et nous obtenons toujours le même type de résultat.

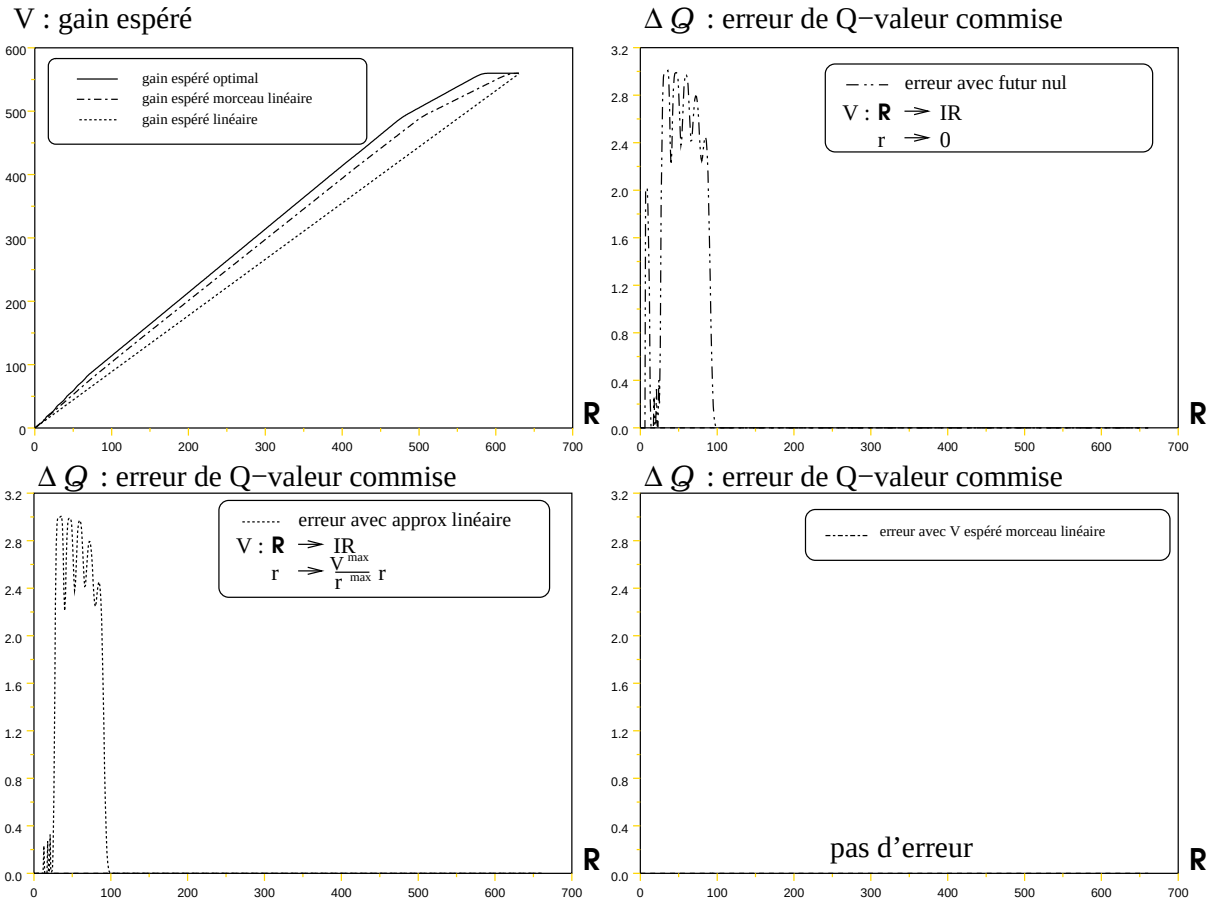


FIG. 11.5 – Une mission composées de nombreuses PRU homogènes mais différentes.

Explications *pour la figure 11.5 : cette mission est composée d'une vingtaine de PRUs de type*[52] 1, 2, 3 *et* 4 *en quantité variables et la PRU locale est de type* 1. *Sur la figure de gauche nous avons représenté le gain espéré pour le reste de la mission obtenu avec trois méthodes. La méthode de gain linéaire par morceaux s'approche plus du gain optimal que la méthode linéaire simple. En haut à droite, nous avons calculé la différence entre les Q-valeurs optimales maximales et les Q-valeurs optimales obtenues en appliquant les actions calculée pour la politique où le futur ne rapporte rien. En dessous, nous présentons les erreurs de Q-valeurs générées par un gain espéré linéaire et notre méthode de recomposition. Les deux méthodes nulle et linéaire pour obtenir une politique approchée génèrent de nombreuses erreurs pour les états pour lesquels il reste*

[51] Ou heureusement...

[52] Ces PRUs sont illustrées en fin de chapitre.

entre 0 *et* 100 *unités de ressource. L'erreur générée par ces méthodes varie autour de* 2.8. *Notre méthode d'approximation linéaire par morceaux ne génère aucune erreur visible sur le graphique. Cependant, nous avons noté que la différence maximale entre les Q-valeurs était de* 2.10^{-3} *alors que la valeur de la qualité accumulable dans une de ces PRUs varie entre* 8 *et* 20. *Cette erreur peut donc à notre sens être négligée.*

Ce résultat est à nos yeux très important : quand les écarts entre les qualités des différents modules à exécuter dans la mission est faible, notre approche fonctionne très bien. Or dans ce cas, les erreurs qui peuvent être commises ne seront jamais très grandes localement[53] par rapport à l'ensemble de la qualité qui pourra ressortir de toute la mission (puisque l'écart-type entre les qualités des modules présents est faible). Notre approche fournit un résultat minutieux, qui tient compte du moindre détail dans la mission, alors qu'une simple approche linéaire permet d'obtenir une politique qui s'éloigne certes peu de l'optimale, mais trop si l'on veut vraiment maximiser la somme totale des qualités accumulées dans la mission en tenant compte des ressources restantes.

L'agent a le même comportement que s'il suivait une politique optimale avec notre approximation. Notons de plus que malgré les nombreux modules présents au premier niveau de notre PRU locale de type 1, l'erreur (en Q-valeur) commise par l'agent est très faible. L'expérience a été renouvelée en faisant varier les qualités des PRUs, en rajoutant et en enlevant des PRUs dans la mission, le résultat reste le même.

On pourrait croire que le faible nombre d'erreurs est seulement du au fait que l'écart-type de la qualité des modules présents dans la mission est faible. Mais, notre méthode d'approximation permet d'obtenir une fonction convexe qui s'approche plus de la fonction optimale que la fonction linéaire $V^{\sim}_{lineaire} : \mathtt{r} \rightarrow \frac{V^{max}}{\mathtt{r}^{max}}.\mathtt{r}$. L'utilisation de la fonction linéaire ne diminue pas significativement le nombre d'erreurs par rapport à l'utilisation de la fonction nulle. En revanche, dès que notre méthode est utilisée, le nombre d'erreurs en Q-valeur diminue en nombre. Les erreurs sont également plus faibles.

Modules bien distincts

Nous obtenons le même résultat lorsque le premier niveau de la PRU courante est composé de deux modules dont l'écart en qualité est grand (mais reste quand même dans la fourchette de l'écart-type qui définit notre homogénéité). Ceci s'explique encore plus facilement que pour le cas précédent, car il est difficile de se tromper de module lorsque ceux-ci sont différents. Dans la plupart des cas, quelle que soit la méthode d'approximation choisie, si les modules sont éloignés, les actions des politiques obtenues sont exactement les mêmes. L'erreur étant nulle, nous ne la représentons pas graphiquement.

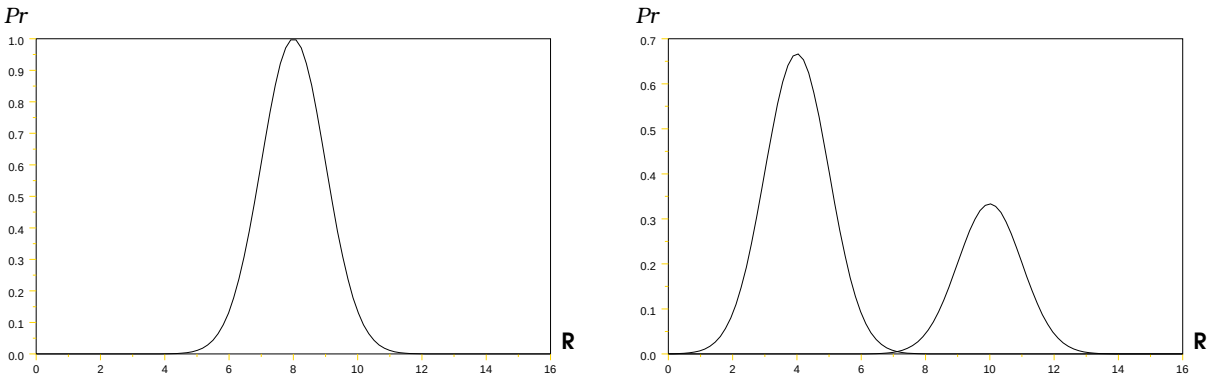
Ensuite, nous avons remarqué que plus le nombre de PRUs est grand, moins le nombre

[53] Les erreurs en Q-valeurs dues à une décision approchée dans la PRU locale.

d'erreurs est grand relativement à l'espace d'état considéré. En fait, plus il y a de ressources et de PRUs moins il y a d'erreurs (ce phénomène se voit bien dans nos expériences, les erreurs sont pour les quantités de ressources faibles). Donc, comme le nombre d'erreurs possibles diminue avec le nombre de PRUs et que le temps pour calculer la politique optimale est de plus en plus grand (croissance exponentielle avec le nombre de PRUs), notre méthode est vraiment efficace pour replanifier rapidement l'exécution d'une PRU locale si les distributions de probabilités de consommation de ressources des modules sont normales.

11.3.4 Mission à distribution bimodale

Dans ce qui précède, nous avons des résultats pour des PRUs où les distribution de probabilités suivent des lois normales. Nous avons conçu des PRUs où les distribution de probabilités de consommation sont bimodales (comme sur la figure 11.6.**b**).



a) loi de distribution normale

b) loi de distribution bimodale

FIG. 11.6 – Loi de distribution normale et bimodale.

Dans la pratique la consommation de ressources d'un robot pour une tâche donnée ne peut pas être bimodale. Nous effectuons ces tests pour savoir si notre algorithme de recomposition permet de retrouver une fonction de valeur proche de l'optimale.

Nous avons calculé les fonctions de gain espéré (approchée et optimale) pour une mission comprenant des modules qui peuvent consommer soit 4 unités de ressources soit 16, rien entre les deux. La qualité qui ressort de ce module est de 900 et tous les autres modules de la mission rapportent une qualité comprise en 2 et 20 et leur distribution de probabilité de consommation de ressources sont normales. Les résultats sont sur la figure 11.7.

Explications *pour la figure 11.7 : nous avons testé notre algorithme de recomposition de gain espéré pour des missions contenant des modules à risques. l'une ou l'autre de nos méthodes de recomposition de fonction de gain espéré (par morceaux de profils et linéaire par morceau) engendre un calcul de politique pour lesquelles beaucoup d'erreurs en Q-valeurs sont commises.*

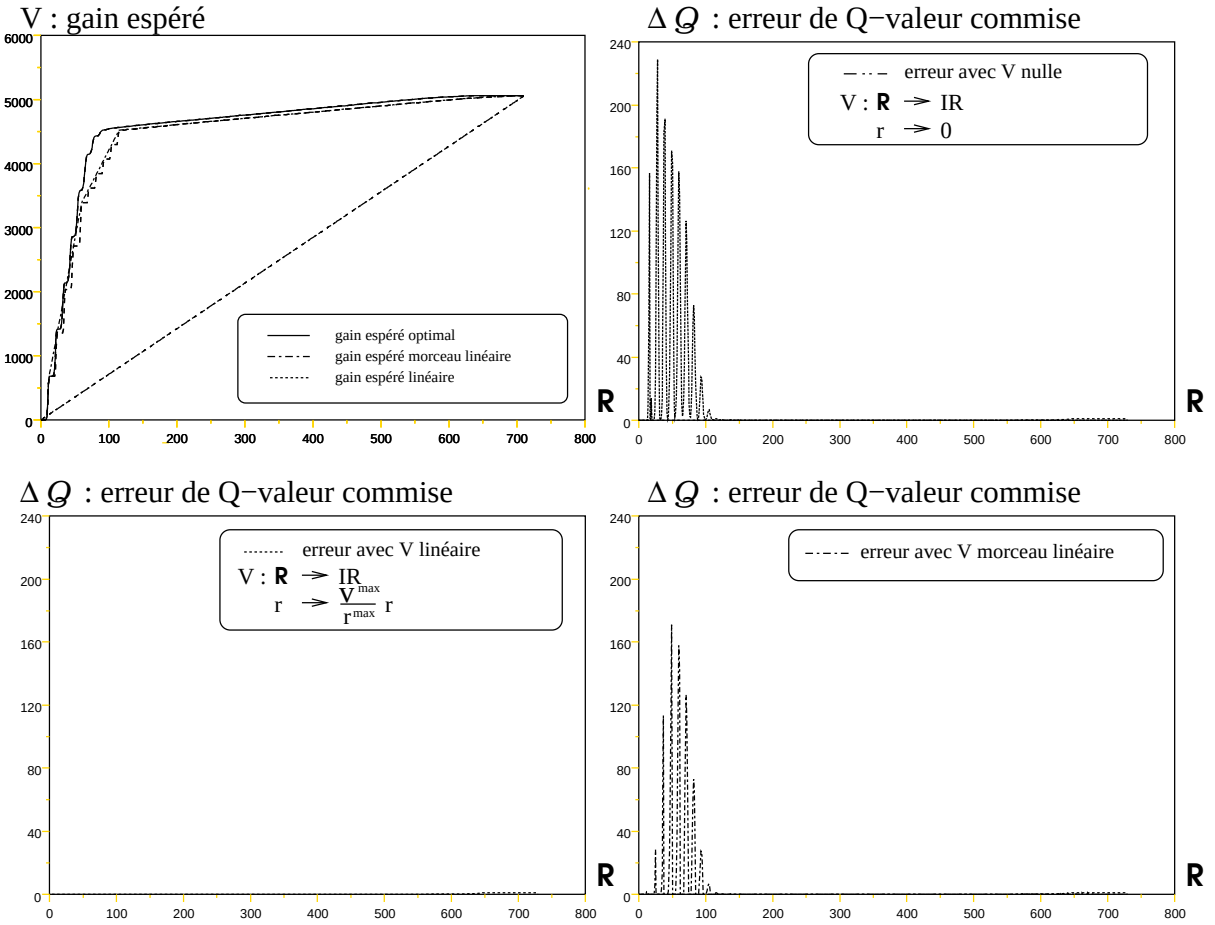


FIG. 11.7 – Un mauvais cas.

Notre fonction d'approximation fonctionne mal dans ce cas. Nous avons un cas où l'approximation linéaire par morceaux surestime le gain espéré V^*, ce qui engendre de très mauvais résultats dans le calcul de la politique. La fonction de gain espéré calculée par ajout de morceaux de profils de performance est très irrégulière et les variations de celles-ci entraînent beaucoup d'erreurs dans les Q-valeurs. Mais outre le fait que de tels modules ne soient pas réalistes, il est, nous semble t-il, normal de trouver des politiques dont l'écart oscille autant puisque les décisions à prendre sont elles aussi risquées. Le gain mesuré par la politique optimale est une mesure qui se base sur une consommation moyenne de ressources et ici, faire la moyenne sur la distribution de probabilité de consommation de ressources n'a pas de sens.

Il est donc important de noter que notre approche ne fonctionne que si les distributions de probabilités sont normales, ce qui est en général le cas dans une application robotique.

Conclusion

Nous avons finalement présenté dans ce chapitre une méthode permettant de valider les algorithmes d'approximations de gain espéré pour une mission donnée. C'est après avoir défini dans le chapitre 7, puis formalisé dans le chapitre 8 un système de contrôle doublement adaptatif permettant à la fois de gérer l'incertitude de consommation de ressources et de pouvoir modifier son comportement en cas de changement dans la mission que nous présentons dans ce chapitre une série de tests qui valide les algorithmes de calculs rapides permettant de fournir immédiatement une politique à suivre en cas de changement dans la mission.

Notre approche se base sur une séparation du calcul de la fonction de gain espéré et du calcul de la politique locale en fonction de gain espéré. Nous avons proposé dans le chapitre 8 un algorithme permettant de calculer rapidement une fonction convexe qui approche la fonction de gain espéré optimale.

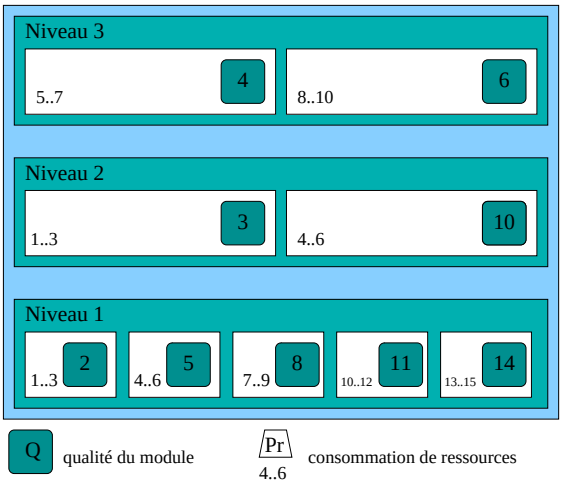
Outre le fait que notre algorithme permet d'accélérer de façon considérable le calcul de la fonction de gain espéré, les valeurs espérées liées aux actions obtenues sont très proches de celles qui auraient été obtenues par programmation dynamique classique. Pour valider la qualité de cette approche, nous avons calculé d'un coté la politique optimale et de l'autre une politique qui se base sur un gain espéré approché, puis comparé les Q-valeurs obtenues (dans la politique optimale) en appliquant dans tous les états soit l'action de la politique optimale, soit l'action de la politique approchée.

Les écarts en Q-valeurs entre les décisions prises pour une politique approchée et une politique optimale sont faibles, voire nuls. Notre approche fonctionne aussi bien qualitativement que la mission soit courte, ou longue.

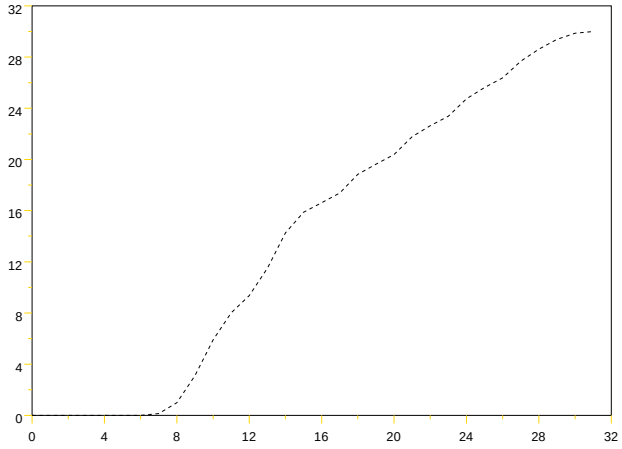
Nous répondons donc bien à l'attente de [Mouaddib et Zilberstein, 2000] : nous avons trouvé un système qui permet de calculer rapidement une politique à adopter dans la PRU locale quand l'ensemble de tâches à effectuer dans le futur peut changer. Mieux même, puisque nous ne faisons aucune hypothèse sur la structure passée des tâches de la mission, nous pouvons établir en quelques millisecondes une politique pour une tâche présente en tenant compte n'importe quelle séquence de tâches futures.

Annexes pour ce chapitre

Ces PRUs ont été utilisées pour les expériences qui sont exposées dans le chapitre. Toutes les PRUs générées ne sont pas mentionnées dans ces annexes, elles sont nombreuses. D'autres PRUs ont été utilisées pour valider les tests, ces PRUs ne sont que des exemples.

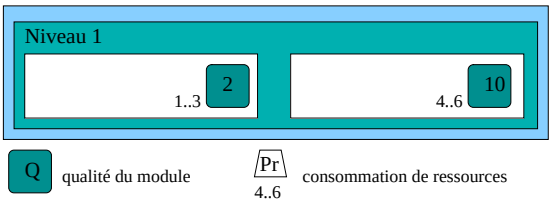


a) la PRU

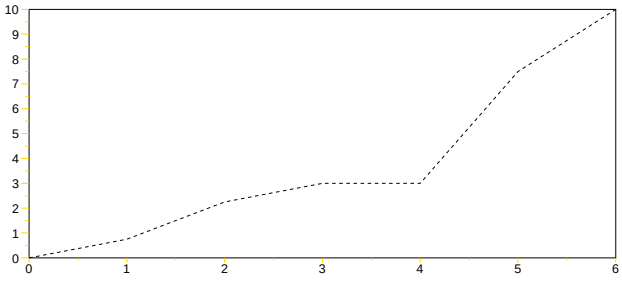


b) profil de performance

FIG. 1 – PRU de type 1.



a) la PRU

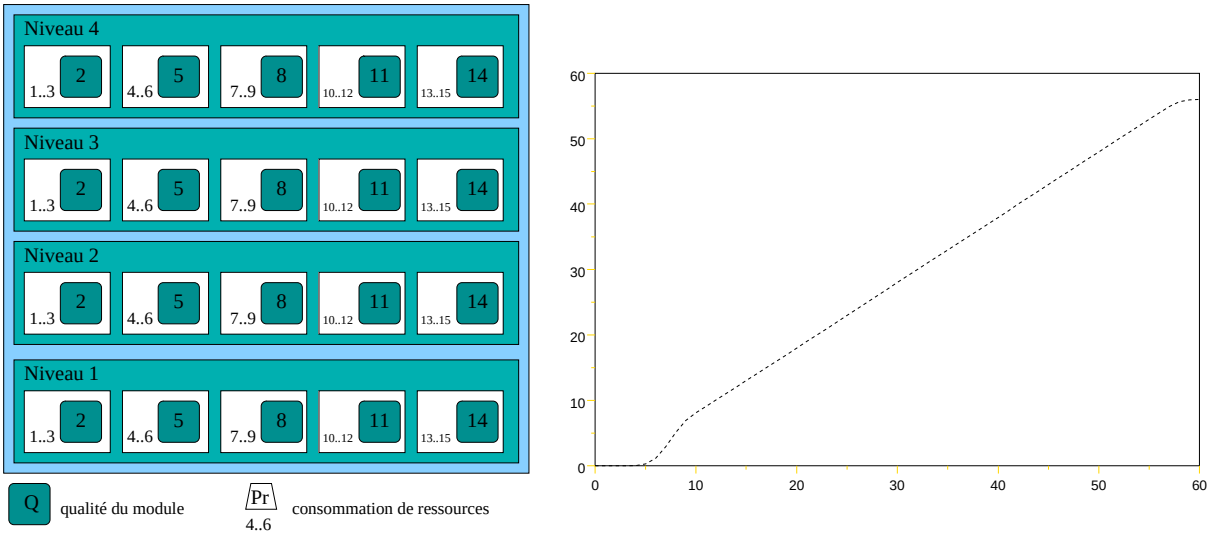


b) profil de performance

FIG. 2 – PRU de type 2.

Ce type de PRU 2 est particulièrement intéressant pour montrer que quand l'écart entre les

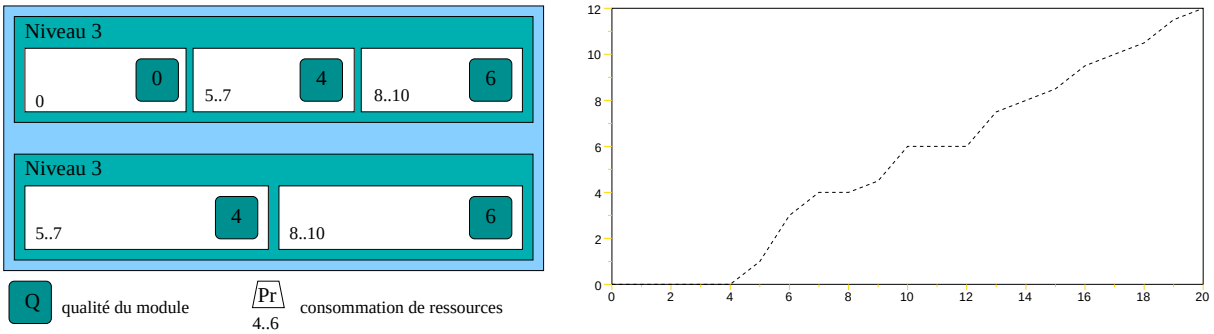
qualités des modules est important, l'agent ne se trompe jamais.



a) la PRU **b)** profil de performance

FIG. 3 – PRU de type 3.

Cette PRU 3 a été utilisée pour montrer que lorsque les modules sont nombreux et proches en qualité, l'agent a une faible perte en Q-valeur en utilisant l'algorithme de recomposition par morceaux.



a) la PRU **b)** profil de performance

FIG. 4 – PRU de type 4.

Conclusion

Cette thèse s'inscrit dans le domaine de l'intelligence artificielle et plus particulièrement dans le domaine de la planification et du contrôle d'agent autonome.

Nous proposons dans cette thèse un mécanisme de contrôle adaptatif permettant à un robot autonome d'effectuer le mieux et le plus possible de tâches complexes tout en gérant les ressources consommables dont il dispose. Nous illustrons ce type de mission dans le premier chapitre par un robot explorateur martien qui doit récolter seul des informations tout au long de la journée avant de transmettre ses résultats vers la Terre.

Dans [Mouaddib et Zilberstein, 2000], les auteurs proposent un système de contrôle adaptatif pour un robot autonome capable d'effectuer ce type de mission. Leur système de contrôle est basé sur le raisonnement progressif, qui fait lui-même partie de l'ensemble des modèles de raisonnement flexibles. Deux questions restent ouvertes :

- leur système de contrôle ne prend en compte qu'une seule ressource consommable, le temps, mais les auteurs envisagent d'étendre le système à plusieurs ressources consommables.
- le système de contrôle permet de gérer les incertitudes liées à la consommation de ressources, mais pas de s'adapter à un environnement dynamique où l'ensemble des tâches complexes de la mission peut changer.

Nous apportons précisément une solution à chacun de ces deux problèmes dans cette thèse.

Bilan

Une mission est un ensemble ordonné de tâches complexes. Il existe plusieurs façons d'exécuter chaque tâche. Ces différentes exécutions consomment plus ou moins de ressources, et produisent, une fois terminées, une qualité sous forme numérique. La consommation de ressources pour effectuer une tâche donnée est incertaine. Disposant d'un robot qui va agir comme un agent rationnel, nous proposons d'utiliser et d'enrichir un système de contrôle qui permet de maximiser la somme totale des qualités qui ressortent des tâches exécutées.

Nous avons présenté dans les chapitres 4 et 5 le raisonnement progressif, structure de contrôle qui a le double avantage de pouvoir modéliser simplement les tâches hiérarchiques, mais aussi de pouvoir calculer à partir de ce modèle un plan qui optimise le comportement de l'agent rationnel. Le critère qu'il devra optimiser est la somme des qualités accumulées lors de la mission mais il devra pour cela utiliser intelligemment ses ressources consommables.

Ce système de contrôle se base sur un problème de décision markovien où les états regroupent les ressources restantes et la tâche en cours d'exécution, les transitions sont liées à la distribution de probabilités de consommation de ressources du modèle, et la récompense est la qualité qui va ressortir de chaque tâche complexe, ou unité de raisonnement progressif (PRU). Le critère à maximiser pour le problème de décision markovien est la somme de toutes les récompenses obtenues pendant la mission. Pour résoudre ce système, on utilise classiquement un algorithme

de programmation dynamique, où chaque état du MDP est évalué une seule fois. Nous obtenons à la suite de ce calcul une politique optimale au sens du problème markovien considéré. Malheureusement, l'espace d'états qu'il faut explorer pour trouver cette politique est grand, et le temps nécessaire pour le calcul croît exponentiellement avec le nombre de PRUs présentes dans la mission. Mais il est possible de séparer le processus de construction de la politique en deux parties : une politique locale, et une fonction de valeur qui estime ce que l'on peut espérer gagner dans le futur.

Cette séparation nous a permis d'étendre le système de contrôle basé sur le raisonnement progressif sur deux points : la gestion de ressources multiples, la replanification rapide de la politique à suivre en cas de changement dans la mission.

Passage aux ressources multiples

Notre premier apport dans cette thèse a été d'étendre le modèle du raisonnement progressif à la gestion de ressources multiples. Pour cela nous avons modifié le modèle des PRUs en introduisant dans les distributions de probabilités de consommation de ressources autant de dimensions qu'il y avait de ressources supplémentaires à gérer. La ressource unique temporelle $\mathtt{t}$ a été transformée en vecteur de ressources $\overline{\mathtt{r}}$ permettant ainsi de tenir compte non seulement du temps qui s'écoule mais aussi de l'énergie consommée : $(\mathtt{t}, \mathtt{e})$.

Le problème introduit par cette extension est connu sous le nom de *malédiction de la dimension*, qui montre d'après [Bellman, 1957] que le temps de calcul de la politique augmente exponentiellement avec le nombre de variables nécessaires pour décrire chaque état. Nous proposons donc un algorithme de calcul de politique qui limite l'explosion de la taille de l'espace d'état à explorer lors de la construction de la fonction de valeur par programmation dynamique, en utilisant d'une part un sens de parcours adéquat de l'espace d'état, et d'autre part un système d'agrégation sur cet espace qui limite le nombre de d'états à stocker et le nombre de calculs à effectuer au total.

La politique obtenue pour la tâche courante grâce au calcul de la fonction de valeur sur l'ensemble des tâches restantes dans la mission est toujours optimale, mais le temps de résolution a été considérablement réduit. Seul un tiers de l'espace total des états possibles est exploré, et pour un même temps d'exécution, nous pouvons maintenant calculer des politiques pour des missions composées de deux fois plus de tâches.

Les résultats que nous présentons sont essentiellement illustrés par un problème avec deux ressources, mais il est possible d'adapter le modèle et les algorithmes à plusieurs ressources pour obtenir également un gain en temps de résolution pour le calcul de la politique qui garantit un comportement optimal vis à vis du critère à optimiser.

Nous avons réussi à contourner partiellement le problème de la malédiction de la dimension,

mais le temps de calcul croît toujours exponentiellement avec le nombre de ressources différentes et au carré avec le nombre de PRUs dans la mission. Nous avons l'intention, et nous le verrons dans la section des perspectives de ce chapitre, d'essayer de trouver un algorithme permettant de trouver la fonction de gain espéré plus rapidement qu'avec la programmation dynamique avec les idées que nous avons exposées pour l'adaptation à un environnement dynamique.

Adaptation à un environnement dynamique

Que se passe-t-il si la séquence de tâches qui constituent la mission est modifiée pendant que le robot l'exécute ? Avec le seul système de contrôle global de raisonnement progressif tel qu'il a été présenté jusqu'à maintenant, il n'est pas possible d'envisager un tel scénario : le temps nécessaire pour calculer une politique optimale pour la tâche courante peut prendre de quelques dizaines de secondes à quelques minutes. C'est un temps qui est largement supérieur à celui qu'il faut pour exécuter un des modules qui constituent les PRUs. Pendant que le module de contrôle recalcule la politique optimale à suivre, le robot ne sait pas quoi faire. Quel module exécuter ?

Nous avons mis au point un système permettant de calculer très rapidement une fonction convexe qui s'approche de la fonction de valeur optimale. Le calcul de cette fonction de gain espéré approchée ne prend que quelques millisecondes, et permet de calculer une politique locale approchée. Le robot peut alors prendre des décisions dans la PRU courante si la séquence de tâches à venir dans la mission est modifiée. Ces décisions ne sont pas forcément celles qu'il aurait prises s'il avait connu la stratégie optimale à adopter, mais faute de mieux, permet d'avoir un bon comportement plutôt que de choisir une action au hasard ou de ne rien faire du tout.

Nous avons comparé les comportements adoptés par le robot qui suit une politique optimale et une politique approchée. Nous avons également comparé les valeurs des décisions prises dans chacun des états possibles pour une nouvelle mission donnée et une PRU courante donnée. Nous avons constaté que le comportement adopté par le robot quand il suit la politique approchée lui fait choisir des actions dont la valeur diffère très peu voire pas du tout de la valeur de l'action qui garantirait un comportement optimal, ceci pour tous les états de la PRU courante. En faisant varier les paramètres de la mission, le nombre et le type de PRUs qui la compose, ainsi que le type de la PRU courante, le résultat reste le même. Notre approximation se base sur une décomposition de la fonction de gain espéré en profils de performance pour chaque tâche qui compose la mission, puis en une recomposition de celle-ci dès qu'un changement intervient dans la mission.

Notre algorithme fonctionne moins bien si la distribution de probabilités de consommation de ressources est éparpillée et discontinue, mais nous avons précisé que ce cas de figure ne correspond pas à une consommation de ressources réaliste pour un robot qui effectue un module, et que le raisonnement progressif n'a pas pour but de contrôler des missions qui présentent de tels risques.

Nous avons donc apporté une solution efficace pour adapter dynamiquement la politique à adopter en cas de changement de dernière seconde dans la mission à effectuer. Le comportement du robot sera presque toujours optimal, même si la mission change.

Perspectives

Notre système de contrôle permet de tenir compte à la fois des incertitudes liées à la consommation de ressources mais aussi de rester efficace en cas de changement dans la séquence de tâches qui composent la mission. Cependant, nous nous sommes limités à la gestion d'une ressource unique dans le cadre des missions pouvant se dérouler dans un environnement dynamique. Nous avons l'intention de mettre en place un système qui combine les deux apports de notre thèse, à savoir la planification dans un environnement dynamique sous contraintes de ressources consommables multiples. Nous avons l'intention de décomposer la fonction de gain espéré avec de multiples ressources avec les profils de performances des différentes tâches qui composent la mission. Cependant, la recomposition va être plus difficile qu'avec une seule ressource, puisqu'il va falloir trouver un système pour allouer les différentes ressources à toutes les tâches et donc se diriger vers le domaine de la décision multicritères.

Une deuxième perspective, sûrement plus rapide à mettre en place serait de créer des modules dont la qualité en sortie ne soit plus déterministe mais probabiliste. En effet, nous considérons pour l'instant qu'une fois un module exécuté, le résultat est toujours aussi satisfaisant, que la qualité est une sortie constante du module. Cependant, on peut imaginer que la qualité varie selon une certaine distribution de probabilités. Ceci entraînerait une explosion de l'espace d'état, mais nous pensons pouvoir la maîtriser en utilisant les mêmes techniques que celles que nous avons présenté pour les ressources multiples, en mettant qualité accumulée et ressources restantes sur le même plan.

Dans un cadre plus général, nous aimerions amener plus de contraintes dans la mission. Des contraintes relatives à l'espace, au temps, et des contraintes de précédences plus fortes entre les tâches.

En ce qui concerne l'espace, nous aimerions pouvoir modéliser les déplacements par des tâches obligatoires dans la mission, qui pourraient aussi être des unités de raisonnement progressif, où l'on pourrait faire varier la vitesse, la consommation d'énergie, la trajectoire... Pour l'instant, dans notre exemple de robot bowling, le compteur temps s'arrête pendant les déplacements du robot, les mouvements ne sont pas comptés dans le plan, et sont artificiellement englobés dans les actions $\mathbf{M}$.

Nous aimerions également introduire des contraintes temporelles plus fortes, des dates butoirs pour les tâches, des dates avant laquelle une tâche ne peut pas commencer, etc... Un des axes principaux de l'équipe MAD du GREYC de Caen dans laquelle nous avons poursuivi notre

thèse est la modélisation du temps ; nous avons bon espoir de pouvoir concilier le raisonnement progressif et les contraintes temporelles et spatiales.

Enfin, une autre de nos perspectives est d'intégrer le raisonnement progressif dans un système multi-agents. Nous avons en effet focalisé nos recherches sur un unique agent autonome, mais il serait semble-t'il intéressant de répartir l'ensemble des tâches d'une mission sur plusieurs agents.

Bibliographie

[Ash *et al.*, 1993] Ash, D., Gold, G., Seiver, A. et Hayes-Roth, B. (1993). Guaranteeing real-time response with limited resources. *Artificial Intelligence in Medicine*, 5(1):49–66.

[Bellman, 1978] Bellman (1978). *An introduction to Artificial intelligence : Can Computers Think ?* Boyd & Fraser Publishing Company, San Francisco.

[Bellman, 1957] Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press.

[Blum et Furst, 1997] Blum, A. et Furst, M. L. (1997). Fast planning through planning graph analysis. *Artif. Intell.*, 90(1-2):281–300.

[Blum et Furst, 1995] Blum, A. L. et Furst, M. L. (1995). Fast planning throught planning-graphs analysis. *In proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, Montreal, Quebec, Canada.

[Boddy, 1991] Boddy, M. (1991). Anytime problem solving using dynamic programming. *In National Conference on Artificial Intelligence (AAAI)*, pages 738–743.

[Boddy et Dean, 1994] Boddy, M. S. et Dean, T. (1994). Deliberation scheduling for problem solving in time-constrained environments. *Artif. Intell.*, 67(2):245–285.

[Boussard *et al.*, 2007] Boussard, M., Mouaddib, A. I. et Bouzid, M. (2007). Towards a formal framework for multi-objective multi-agent planning. *In AAMAS*.

[Boutilier *et al.*, 1998] Boutilier, C., Brafman, R. I. et Geib, C. (1998). Strutured reachability analysis for markov decision processes. *In proceedings of the 14th Conference on Uncertainty in Artificial Intelligence, Madison, WI*, pages 24–32.

[Boutilier *et al.*, 2000] Boutilier, C., Dearden, R. et Goldszmidt, M. (2000). Stochastic dynamic programming with factored representations. *Artif. Intell.*, 121(1-2):49–107.

[Cardon *et al.*, 2001] Cardon, S., Mouaddib, A. I., Zilberstein, S. et R.Washington (2001). Adaptive control of acyclic progressive processing task structures. *In proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 701–706, Seattle, Washington, USA.

[Chapman, 1987] Chapman, D. (1987). Planning for conjunctive goals. *Artif. Intell.*, 32(3):333–377.

[Charniak et McDermott, 1985] Charniak et McDermott (1985). *Introduction to Artificial Intelligence.* Addison-Wesley, Reading,Massachussets.

[Dean et Boddy, 1988] Dean, T. et Boddy, M. (1988). An analysis of time-dependant planning. *In National Conference on Artificial Intelligence (AAAI)*, pages 49–54.

[Dean et Givan, 1997] Dean, T. et Givan, R. (1997). Model minimization in markov decision processes. *In AAAI/IAAI*, pages 106–111.

[Dean et Lin, 1995] Dean, T. et Lin, S. H. (1995). Decomposition techniques for planning in stochastic domains. *In proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1121–1127, Montreal, Quebec, Canada.

[Dearden et Boutilier, 1997] Dearden, R. et Boutilier, C. (1997). Abstraction and approximate decision-theoretic planning. *Artif. Intell.*, 89(1-2):219–283.

[Decker et Lesser, 1993] Decker, K. et Lesser, V. (1993). Qualitative modeling of complex computationnal task environment. *In National Conference on Artificial Intelligence (AAAI)*, pages 217–224.

[Feng *et al.*, 2004] Feng, Z., Dearden, R., Meuleau, N. et Washington, R. (2004). Dynamic programming for structured continuous markov decision problems. *In proceedings of UAI 2004*, pages 154–161.

[Ferber, 1995] Ferber, J. (1995). *Les Systèmes multi-agents : Vers une intelligence collective.* Dunod.

[Fikes et Nilsson, 1971] Fikes, R. et Nilsson, N. J. (1971). Strips : A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208.

[Garvey et Lesser, 1993] Garvey, A. et Lesser, V. (1993). Design-to-time real-time scheduling. *In IEEE Transaction on systems, man, cybernetics, 23(6)*, pages 1491–1502.

[Ghallab *et al.*, 2004] Ghallab, M., Nau, D. et Traverso, P. (2004). *Automated Planning : Theory and Practice.* Morgan Kaufmann, San Francisco, CA.

[Green, 1969] Green, C. (1969). Application of thorem proving to planning. *In proceedings of the first International Joint Conference on Artificial Intelligence (IJCAI)*, pages 219–239.

[Hanks, 1993] Hanks, S. (1993). *Projecting plans for uncertain worlds.* Thèse de doctorat, Yale University, Departement of Computer Science, New Haven, CT.

[Hanks et McDermott, 1994] Hanks, S. et McDermott, D. V. (1994). Modeling a dynamic and uncertain world i : Symbolic and probabilistic reasoning about change. *Artif. Intell.*, 66(1):1–55.

[Harlemen et Teije, 2000] Harlemen, V. et Teije, A. (2000). Anytime diagnostic reasoning using approximate boolean constraint propagation. *In Proceedings of the Seventh Interna-*

tional Conference on Principles of Knowledge Representation and Reasoning (KR2000), Breckenridge, Colorado, USA.

[Haugeland, 1985] Haugeland (1985). *Artificial Intelligence : the Very Idea*. MIT Press, Cambridge, Massachussets.

[Hayes-Roth, 1990] Hayes-Roth, B. (1990). Architectural foundation for real times performance in intelligent agents. *Journal of Real-Time Systems*, 0(0):99–125.

[Howard, 1960] Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. MIT Press, Cambridge Massachussetts.

[Howe *et al.*, 1990] Howe, A., Hart, D. et Cohen, P. (1990). Addressing real-time constraints in the design of autonomous agents. *Journal of Real-Time Systems*, 2(1-2):81–97.

[Hundt *et al.*, 2006] Hundt, C., Panangaden, P., Pineau, J. et Precup, D. (2006). Representing systems with hidden state. *In proceedings of the Twenty-First National Conference of Artificial Intelligence (AAAI)*.

[Knoblock, 1994] Knoblock, C. A. (1994). Automatically generating abstractions for planning. *Artif. Intell.*, 68(2):243–302.

[Korf, 1987] Korf, R. E. (1987). Real time heuristics search : first results. *In National Conference on Artificial Intelligence (AAAI)*, pages 133–138.

[Kurzweil, 1990] Kurzweil (1990). *The Age of Intelligent Machines*. MIT Press, Cambridge, Massachussets.

[Kushmerick *et al.*, 1995] Kushmerick, N., Hanks, S. et Weld, D. S. (1995). An algorithm for probabilistic planning. *Artif. Intell.*, 76(1-2):239–286.

[Kveton et Hauskrecht, 2006] Kveton, B. et Hauskrecht, M. (2006). Solving factored mdps with exponential-family transition models. *In In proceedings of the 16th International Conference on Planning and Scheduling, UK*.

[Le Gloannec *et al.*, 2004] Le Gloannec, S., Mouaddib, A. I. et Charpillet, F. (2004). Planning under uncertainty with multiple consumable resources. *In proceedings of Workshop on Agents in dynamic and Real Time Environments, in the 16th European Conference on Artificial Intelligence, (ECAI)*.

[Le Gloannec *et al.*, 2005a] Le Gloannec, S., Mouaddib, A. I. et Charpillet, F. (2005a). A decision-theoretic scheduling of resource-bounded agents in dynamic environments. *In proceedings of 13th International Conference on Automated Planning&Scheduling (ICAPS), Poster Session*.

[Le Gloannec *et al.*, 2005b] Le Gloannec, S., Mouaddib, A. I. et Charpillet, F. (2005b). Meta-level control under uncertainty for handling multiple consumable resources of robots. *In proceedings of (IROS)*.

[Lin, 1997] Lin, S. (1997). *Exploiting structure for Planning and Control.* Thèse de doctorat, Brown University.

[Littman, 1996] Littman, M. L. (1996). *Algorithms for Sequential Decision Making.* Thèse de doctorat, Computer Science Department, Brown University.

[Loth *et al.*, 2007] Loth, M., Davy, M. et Preux, P. (2007). Sparse temporal difference learning using LASSO. *In IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning.*

[Meuleau *et al.*, 1998] Meuleau, N., Hauskrecht, M., Kim, K., Peshkin, L., Kealbling, L., Dean, T. et Boutilier, C. (1998). Solving very large weakly coupled markov decision processes. *In proceedings of the 14th Conference on Uncertainty in Artificial Intelligence, Madison, WI.*

[Mouaddib, 1993] Mouaddib, A. I. (1993). *Contribution au raisonnement progressif et temps réel dans un univers multi-agents.* Thèse de doctorat, Université de Nancy I.

[Mouaddib, 2004] Mouaddib, A. I. (2004). Multi-objective decision-theoretic path planning. *In IEEE International Conference on Robotics and Automaton (ICRA), New Orleans, USA,*, pages –.

[Mouaddib, 2006] Mouaddib, A. I. (2006). Collective multi-objective planning. *In IEEE Distributed Intelligent Systems : Collective Intelligence and its applications, Prague, Republique tcheque*, pages –.

[Mouaddib *et al.*, 1994] Mouaddib, A.-I., Charpillet, F. et Haton, J. (1994). GREAT : a model of progressive reasoning for real-time systems. *In IEEE International Conference on Tools for AI (ICTAI)*, pages 521–527.

[Mouaddib et Le Gloannec, 2006] Mouaddib, A. I. et Le Gloannec, S. (2006). A cooperative distributed problem solving technique for large markov decision processes. *In proceedings of the 17th proceedings of European Conference on Artificial Intelligence (ECAI), Poster session.*

[Mouaddib et Zilberstein, 1995] Mouaddib, A.-I. et Zilberstein, S. (1995). Knowledge-based anytime computation. *In proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 775–783.

[Mouaddib et Zilberstein, 1997] Mouaddib, A. I. et Zilberstein, S. (1997). Handling duration uncertainty in meta-level control of progressive processing. *In proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1201–1206, Nagoya, Japan.

[Mouaddib et Zilberstein, 1998] Mouaddib, A. I. et Zilberstein, S. (1998). Optimal scheduling for dynamic progressive processing. *In proceedings of the 15th European Conference on Artificial Intelligence (ECAI)*, pages 499–503.

[Mouaddib et Zilberstein, 1999] Mouaddib, A. I. et Zilberstein, S. (1999). Reactive control of dynamic progressive processing. *In proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, Stockholm, Sweden.

[Mouaddib et Zilberstein, 2000] Mouaddib, A. I. et Zilberstein, S. (2000). Optimizing resource utilization in planetary rovers. *In proceedings of the 2nd NASA International Workshop on Planning and Scheduling for Space*, pages 163–168.

[Musliner *et al.*, 1993] Musliner, D., Durfee, E. et Shin, K. (1993). Circa : a cooperative intelligent real-time control archictecture. *In IEEE Transactions on Systems, Man and cybernetics.*

[Newell et Simon, 1963] Newell, A. et Simon, H. A. (1963). Gps, a program that simulates human thought. *In Computers and Thought, MacGrawHill, New York.*

[Newell et Simon, 1972] Newell, A. et Simon, H. A. (1972). *Human Problem Solving.* Prentice Hall, EngleWood Cliffs, NJ.

[Nilsson, 1998] Nilsson (1998). *Artificial Intelligence : A New Synthesis.* Morgan Kaufmann, San Mateo, California.

[Parr, 1998] Parr, R. (1998). Flexible decomposition algorithms for weakly coupled markov decision process. *In proceedings of the 14th Conference on Uncertainty in Artificial Intelligence, Madison, WI.*

[Pearl, 1983] Pearl, J. (1983). *Heuristics.* Morgan Kaufmann, San Mateo, CA.

[Pearl, 1988] Pearl, J. (1988). *Probabilistic Reasonning in Intelligents Systems : Network of Plausible Inference.* Morgan Kaufmann, San Mateo, CA.

[Pineau *et al.*, 2003] Pineau, J., Gordon, G. J. et Thrun, S. (2003). Point-based value iteration : An anytime algorithm for pomdps. *In proceedings of the 18th International Joint Conference on Artificial Intelligence IJCAI*, pages 1025–1032.

[Poole *et al.*, 1998] Poole, D., Mackworth, A. et Goebel, R. (1998). *Computationnal Intelligence : A logical approach.* Oxford University Press, Oxford UK.

[Pos, 1993] Pos, A. (1993). *Time-Constrained Model-Based Diagnosis.* Thèse de doctorat, University of Twente, the Netherland.

[Preux, 2004] Preux, P. (2004). Propagation of Q-values in tabular TD(λ). *Lecture Notes in Computer Science*, 2430/2002:47–76.

[Puterman, 1994] Puterman, M. L. (1994). *Markov decision processes.* John Wiley and Sons, INC.

[Rich et Knight, 1991] Rich, E. et Knight, K. (1991). *Artificial Intelligence (Second edition).* McGraw-Hill, New York.

[Russel et Norvig, 2003] Russel, S. et Norvig, P. (2003). *Artificial Intelligence : A Modern Approach, second edition.* Prentice Hall.

[Sabbadin, 2002] Sabbadin, R. (2002). Graph partitioning techniques for markov decision processes decomposition. *In ECAI*, pages 670–674.

[Sacerdoti, 1975] Sacerdoti, E. D. (1975). The non-linear nature of plans. *In proceedings of the 4th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 206–214, Tbilisi, USSR.

[Simon, 1996] Simon, H. (1996). *The Sciences of the Artificial.* MIT Press, Cambridge, Massachussets.

[Singh et Cohn, 1998] Singh, S. P. et Cohn, D. (1998). How to dynamically merge markov decision processes. *In Advances in Neural Information Processing Systems 10*, pages 1057–1063.

[Sussman, 1974] Sussman, G. J. (1974). The virtuous nature of bugs. *In proceedings of the first conference of the Society for the Study of AI and the Simulation of Behaviour.*

[Teichteil, 2004] Teichteil, F. (2004). *Approche Symbolique et Heuristique de la planification en Environnement incertain : Optimisation d'une Stratégie de Déplacement de de Prise d'Information.* Thèse de doctorat, École Nationale Supérieure de l'Aéronautique et de l'Espace.

[Turner, 2005] Turner, R. M. (2005). Intelligent mission planning and control of autonomous underwater vehicles. *In proceedings of Workshop on Planning under Uncertainty for Autonomous Systems, in 13th International Conference on Automated Planning&Scheduling (ICAPS)*, Monterey, CA, US.

[Winston, 1992] Winston (1992). *Artificial Intelligence (Third edition).* Addison-Wesley, Reading,Massachussets.

[Zilberstein, 1993] Zilberstein, S. (1993). *Operational Rationality through Compilation of Anytime Algorithms.* Thèse de doctorat, University of California at Berkeley.

[Zilberstein, 1996] Zilberstein, S. (1996). Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83.

[Zilberstein *et al.*, 2000] Zilberstein, S., Mouaddib, A. I. et Arnt, A. (2000). Dynamic scheduling of progressive processing plans. *In ECAI Workshop on New Results in Planning, Scheduling and Design.*

[Zilberstein et Russell, 1996] Zilberstein, S. et Russell, S. J. (1996). Optimal composition of real-time systems. *Artif. Intell.*, 82(1-2):181–213.