PLANIFICATION D'ACTIONS CONCURRENTES SOUS CONTRAINTES ET INCERTITUDE

par

Éric Beaudry

Thèse présentée au Département d'informatique
en vue de l'obtention du grade de philosophiæ doctor (Ph.D.)

FACULTÉ DES SCIENCES

UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, 23 mars 2011

Sommaire

Cette thèse présente des contributions dans le domaine de la planification en intelligence artificielle, et ce, plus particulièrement pour une classe de problèmes qui combinent des actions concurrentes (simultanées) et de l'incertitude. Deux formes d'incertitude sont prises en charge, soit sur la durée des actions et sur leurs effets. Cette classe de problèmes est motivée par plusieurs applications réelles dont la robotique mobile, les jeux et les systèmes d'aide à la décision. Cette classe a notamment été identifiée par la NASA pour la planification des activités des rovers déployés sur Mars.

Les algorithmes de planification présentés dans cette thèse exploitent une nouvelle représentation compacte d'états afin de réduire significativement l'espace de recherche. Des variables aléatoires continues sont utilisées pour modéliser l'incertitude sur le temps. Un réseau bayésien, qui est généré dynamiquement, modélise les dépendances entre les variables aléatoires et estime la qualité et la probabilité de succès des plans. Un premier planificateur, ActuPlan^{nc} basé sur un algorithme de recherche à chaînage avant, prend en charge des actions ayant des durées probabilistes. Ce dernier génère des plans non conditionnels qui satisfont à une contrainte sur la probabilité de succès souhaitée. Un deuxième planificateur, ActuPlan, fusionne des plans non conditionnels afin de construire des plans conditionnels plus efficaces. Un troisième planificateur, nommé QuanPlan, prend également en charge l'incertitude sur les effets des actions. Afin de modéliser l'exécution simultanée d'actions aux effets indéterminés, QuanPlan s'inspire de la mécanique quantique où des états quantiques sont des superpositions d'états classiques. Un processus décisionnel de Markov (MDP) est utilisé pour générer des plans dans un espace d'états quantiques. L'optimalité, la complétude, ainsi que les limites de ces planificateurs sont discutées. Des

comparaisons avec d'autres planificateurs ciblant des classes de problèmes similaires démontrent l'efficacité des méthodes présentées. Enfin, des contributions complémentaires aux domaines des jeux et de la planification de trajectoires sont également présentées.

Mots-clés: Intelligence artificielle ; planification ; actions concurrentes ; incertitude.

Préface

Les travaux présentés dans cette thèse n'auraient pas été possibles sans de nombreux appuis. Je veux tout d'abord remercier Froduald Kabanza et François Michaud qui ont accepté de diriger mes travaux de recherche. Leur grande disponibilité et leur enthousiasme témoignent de leur confiance en la réussite de mes travaux et ont été une importante source de motivation.

Je tiens aussi à souligner la contribution de plusieurs amis et collègues du laboratoire Planiart au Département d'informatique de la Faculté des sciences. En plus d'être des co-auteurs de plusieurs articles, Francis Bisson et Simon Chamberland m'ont grandement aidé dans la révision de plusieurs de mes articles. Je tiens également à remercier Jean-François Landry, Khaled Belgith, Khalid Djado et Mathieu Beaudoin pour leurs conseils, leur soutien, ainsi que pour les bons moments passés en leur compagnie. Je remercie mes amis et collègues du laboratoire IntRoLab à la Faculté de génie, dont Lionel Clavien, François Ferland et Dominic Létourneau avec qui j'ai collaboré sur divers projets au cours des dernières années.

Je tiens également à remercier mes parents, Gilles et Francine, ainsi que mes deux frères, Pascal et Alexandre, qui m'ont toujours encouragé dans les projets que j'ai entrepris. Je remercie mon amie de cœur, Kim Champagne, qui m'a grandement encouragé au cours de la dernière année. Je souligne aussi l'appui moral de mes amis proches qui se sont intéressés à mes travaux.

Cette thèse a été possible grâce au soutien financier du Conseil de la recherche en sciences naturelles et génie du Canada (CRSNG) et du Fonds québécois de la recherche sur la nature et les technologies (FQRNT).

Les parties en français privilégient l'usage de la graphie rectifiée (recommandations de 1990) à l'exception de certains termes ayant une graphie similaire en anglais.

Abréviations

AI *Artificial Intelligence*

AAAI *Association for the Advance of Artificial Intelligence*

BN Réseaux bayésien (*Bayesian Netowrk*)

CoMDP Processus décisionnel markovien concurrent (*Concurrent Markov Decision Process*)

CPTP *Concurrent Probabilistic Temporal Planning*

FPG *Factored Policy Gradient [Planner]*

GTD *Generate, Test and Debug*

IA Intelligence artificielle

ICR Centre instantanné de rotation (*Instantaneous Center of Rotation*)

ICAPS *International Conference on Automated Planning and Scheduling*

LRTDP *Labeled Real-Time Dynamic Programming*

MDP Processus décisionnel markovien (*Markov Decision Process*)

NASA *National Aeronautics and Space Administration* des États-Unis d'Amérique

PDDL *Planning Domain Definition Language*

PDF Fonction de densité de probabilité (*Probability Density Function*)

RTDP *Real-Time Dynamic Programming*

Table des matières

Liste des figures

Liste des tableaux

Introduction

La prise de décision automatique représente une capacité fondamentale en intelligence artificielle (IA). Cette capacité est indispensable dans de nombreux systèmes intelligents devant agir de façon autonome, c'est-à-dire sans interventions externes, qu'elles soient humaines ou d'autres natures. Par exemple, un robot mobile doit prendre une multitude de décisions afin d'accomplir sa mission. Ses décisions peuvent se situer à plusieurs niveaux, comme de sélectionner sa prochaine tâche, de choisir sa prochaine destination, de trouver un chemin sécuritaire, et d'activer ses actionneurs et ses capteurs. De façon similaire, les personnages animés dans les jeux vidéos doivent également adopter automatiquement des comportements qui contribuent à augmenter le réalisme du jeu, et ce, dans l'ultime but d'agrémenter l'expérience de jeu des joueurs humains. D'autres applications, comme des systèmes d'aide à la prise de décisions, doivent proposer des actions et parfois même les justifier à l'aide d'explications concises.

Fondamentalement, une décision implique le choix d'une action à prendre. Tout comme les humains qui sont responsables de leurs choix, donc de leurs agissements, un agent intelligent est lui aussi responsable de ses décisions, donc de ses actions. Cette lourde responsabilité implique le besoin d'évaluer et de raisonner sur les conséquences de ses actions. Ce raisonnement est indispensable puisque les conséquences d'une action peuvent avoir des implications considérables sur d'autres actions futures. Cela est d'autant plus important lorsque des actions ont des conséquences subtiles qui peuvent retirer des possibilités à l'agent de façon irréversible, ou impliquer des couts significatifs. Le problème décisionnel devient nettement plus complexe lorsque les conséquences des actions sont incertaines.

Selon la complexité de la mission à accomplir, un agent intelligent peut disposer de

plusieurs options, c'est-à-dire différentes façons d'agencer ses actions au fil du temps. Dans ce contexte précis, une option est fondamentalement un plan d'actions. De façon générale, l'existence de plusieurs options implique la capacité de les simuler à l'avance afin de retenir la meilleure option possible. Ainsi, le problème de prise de décisions automatique peut être vu comme un problème de planification où le meilleur plan d'actions est recherché. En d'autres mots, un agent intelligent doit soigneusement planifier ses actions afin d'agir de façon rationnelle.

Puisque la planification nécessite de connaitre les conséquences des actions planifiées, un agent intelligent doit disposer d'un modèle de lui-même et de l'environnement dans lequel il évolue. Le monde réel étant d'une immense complexité, un modèle fidèle à la réalité est généralement hors de portée en raison des ressources limitées en capacité de calcul et en mémoire. Ainsi, des simplifications dans la modélisation sont incontournables. Ces simplifications se font à l'aide de différentes hypothèses de travail qui réduisent la complexité des problèmes de planification. Cela permet de trouver des plans dans des délais raisonnables, donc de prendre plus rapidement des décisions. En contrepartie, les hypothèses simplificatrices adoptées peuvent affecter, à la baisse, la qualité des décisions prises. Sans s'y limiter, les hypothèses couramment utilisées [51] sont :

- *Observabilité totale*. À tout instant, tout ce qui est requis d'être connu sur le monde (l'environnement) [1] est connu. Par exemple, dans un domaine robotique, la position du robot pourrait être réputée comme étant parfaitement connue à tout instant.
- *Déterministe*. Le résultat d'une action est unique et constant. En d'autres mots, il est présumé que l'exécution se déroule dans un monde parfait où les actions ne peuvent pas échouer et que leurs effets sont totalement prédéterminés.
- *Monde statique*. Il n'y a pas d'évènements externes qui modifient le monde.
- *Plans séquentiels*. Les plans sont des séquences d'actions où chaque action s'exécute l'une à la suite de l'autre. Il n'y a pas d'actions concurrentes (simultanées).
- *Durée implicite*. Les actions n'ont pas de durée, elles sont considérées comme

1. Dans cette thèse, les mots *monde* et *environnement* sont des quasi-synonymes. Dans la littérature, le mot *monde* (*world* en anglais) est généralement employé en planification en IA pour désigner une simplification (un modèle) de ce qui est appelé *environnement* en IA et en robotique mobile.

étant instantanées ou de durée unitaire au moment de la planification.

En combinant les hypothèses précédentes, plusieurs classes de problèmes (et d'algorithmes) de planification sont créées : classique, déterministe, non déterministe, probabiliste, etc. Les frontières entre ces classes ne sont pas toujours nettes et certaines se chevauchent. Chaque classe admet un ensemble de domaines de planification.

Essentiellement, les travaux de recherche en planification en IA consistent à concevoir une solution, soit une combinaison d'algorithmes de planification, d'heuristiques et diverses stratégies, pour résoudre des problèmes d'une classe donnée. Cette solution est intimement liée à un ensemble très précis d'hypothèses simplificatrices qui sont soigneusement présélectionnées en fonction des applications ciblées. Le but est de trouver un juste compromis entre la qualité des décisions prises et les ressources nécessaires en temps de calcul et en mémoire. Ce défi est d'autant plus ambitieux considérant qu'une solution générale est hautement souhaitable. En d'autres mots, on désire une solution de planification qui est la plus indépendante possible de l'application ciblée afin qu'elle soit facilement adaptable à de nouvelles applications.

Au cours des dernières années, des progrès considérables ont été réalisés dans le domaine de la planification en IA. Pour certains types de problèmes de planification, des planificateurs sont capables de générer efficacement des plans comprenant des centaines ou même des milliers d'actions. Par contre, ces planificateurs sont souvent basés sur des hypothèses peu réalistes ou tout simplement trop contraignantes pour être utilisés dans des applications réelles.

Cette thèse présente des avancées pour une classe de problèmes de planification spécifique, soit celle qui combine des actions concurrentes (simultanées) et de l'incertitude. L'incertitude peut se manifester par différentes formes, comme sur la durée des actions, la consommation et la production de ressources (ex. : énergie), et sur les effets des actions tels que leur réussite ou échec. Par son immense complexité, cette classe présente un important défi [17]. Ce type de problèmes est motivé par plusieurs applications concrètes.

Un premier exemple d'application est la planification de tâches pour des robots évoluant dans des environnements où des humains sont présents. Ces robots doivent être capables d'effectuer des actions simultanées, comme de se déplacer tout en effectuant d'autres tâches. Ils font face à différentes incertitudes qui sont liées à la

dynamique de l'environnement et à la présence d'humains. Par exemple, suite aux participations aux *AAAI Robot Challenge* de 2005 et 2006 [49], une difficulté observée fut la navigation dans des foules. Les capteurs étant obstrués, le robot avait de la difficulté à se localiser. En plus d'avoir à contourner des personnes, qui représentent des obstacles mobiles, la difficulté à se localiser a rendu la vitesse de déplacement très imprévisible. Les interactions humain-robot ont également causé des difficultés puisqu'elles ont aussi des durées incertaines. À cette époque, le robot utilisait un planificateur déterministe [8] intégré dans une architecture de planification réactive [7]. Un paramètre important du modèle du planificateur était la vitesse du robot. Une valeur trop optimiste (grande) pouvait entrainer des échecs au niveau du non-respect des contraintes temporelles, alors qu'une valeur trop prudente (petite) pouvait pouvait occasionner le manque d'opportunités. Un planificateur considérant l'incertitude sur la durée des actions pourrait améliorer la performance d'un tel robot.

Un deuxième exemple d'application associée à cette classe est la planification des activités des robots (*rovers*) déployés sur Mars [17]. Les robots *Spirit* et *Opportunity* déployés par la NASA doivent effectuer des déplacements, préchauffer et initialiser leurs instruments de mesure, et acquérir des données et des images en plus et les transmettre vers la Terre. Pour augmenter leur efficacité, ces robots peuvent exécuter plusieurs actions simultanément. Ces robots sont sujets à différentes sources d'incertitudes. Par exemple, la durée et l'énergie requises pour les déplacements sont incertaines. Les données acquises par les capteurs ont également une taille incertaine, ce qui a des impacts sur la durée des téléchargements. Actuellement, les tâches de ces robots sont soigneusement planifiées à l'aide de systèmes de planification à initiatives mixtes [1] où un programme guide un utilisateur humain dans la confection des plans. Des planificateurs plus sophistiqués pourraient accroitre le rendement de ces robots.

Les jeux vidéos représentent un troisième exemple d'application pouvant contenir des actions concurrentes et de l'incertitude. L'IA dans les jeux peut avoir à commander simultanément un groupe d'agents devant se comporter de façon coordonnée. L'incertitude peut également se manifester de différentes façons dans les jeux. Par exemple, le hasard est omniprésent dans les jeux de cartes et dans les jeux de plateau impliquant des lancers de dés. Les jeux de stratégies peuvent également contenir des actions ayant des effets stochastiques. Cela s'ajoute à une autre dimension importante

des jeux, soit la présence d'un ou plusieurs adversaires. Il ne s'agit plus uniquement d'évaluer la conséquence de ses propres actions, mais également d'être en mesure de contrer les actions de l'adversaire.

Les systèmes d'aide à la décision sont d'autres applications pouvant contenir des actions simultanées avec des effets probabilistes. CORALS [13, 14] est un exemple de système d'aide à la décision, développé pour la division Recherche et développement pour la défense Canada, qui supportera les opérateurs de navires déployés dans des régions dangereuses dans le cadre de missions humanitaires ou de maintien de la paix [2]. Les attaques ennemies pouvant être rapides et coordonnées, les opérateurs doivent réagir rapidement en utilisant plusieurs ressources de combats simultanément. Les actions sont également sujettes à différentes formes d'incertitude, la principale étant le taux de succès des armes qui dépend de plusieurs facteurs.

Les contributions présentées dans cette thèse visent à améliorer la qualité et la rapidité de la prise de décisions pour des applications combinant des actions concurrentes sous incertitude. La thèse est composée de quatre chapitres, chacun présentant des contributions spécifiques.

Le chapitre 1 présente le planificateur ActuPlan qui permet de résoudre des problèmes de planification avec des actions concurrentes, et de l'incertitude et des contraintes sur le temps et les ressources. Une représentation d'états basée sur un modèle continu du temps et des ressources est présentée. Contrairement aux approches existantes, cela permet d'éviter une explosion de l'espace d'états. L'incertitude sur le temps et les ressources est modélisée à l'aide de variables aléatoires continues. Un réseau bayésien construit dynamiquement [3] permet de modéliser la relation entre les différentes variables. Des requêtes d'inférences dans ce dernier permettent d'estimer la probabilité de succès et la qualité des plans générés. Un premier planificateur, ActuPlan^{nc}, utilise un algorithme de recherche à chainage avant pour générer efficacement des plans non conditionnels qui sont quasi optimaux. Des plans non conditionnels sont des plans qui ne contiennent pas de branchements conditionnels qui modifient le déroulement de l'exécution selon la durée effective des actions.

2. L'auteur de la thèse a participé au développement des algorithmes de planification du système CORALS, mais à l'extérieur du doctorat.

3. À ne pas confondre avec un réseau bayésien dynamique.

ActuPlan^{nc} est ensuite adapté pour générer plusieurs plans qui sont fusionnés en un plan conditionnel potentiellement meilleur. Les plans conditionnels sont construits en introduisant des branchements conditionnels qui modifie le déroulement de l'exécution des plans selon des observations à l'exécution. La principale observation considérée est la durée effective des actions exécutées. Par exemple, lorsque des actions s'exécutent plus rapidement que prévu, il devient possible de saisir des opportunités en modifiant les façon de réaliser le reste du plan.

Le chapitre 2 présente le planificateur QuanPlan. Il s'agit d'une généralisation à plusieurs formes d'incertitude. En plus de gérer l'incertitude sur le temps et les ressources, les effets incertains des actions sont également considérés. Une solution hybride, qui combine deux formalismes bien établis en IA, est proposée. Un réseau bayésien est encore utilisé pour prendre en charge l'incertitude sur la durée des actions tandis qu'un processus décisionnel markovien (MDP) s'occupe de l'incertitude sur les effets des actions. Afin de résoudre le défi de la concurrence d'actions sous incertitude, l'approche s'inspire des principes de la physique quantique. En effet, QuanPlan effectue une recherche dans un espace d'états quantiques qui permet de modéliser des superpositions indéterminées d'états.

Le chapitre 3 explore une application différente où la concurrence d'actions se manifeste dans une situation d'adversité. L'IA devient un élément de plus en plus incontournable dans les jeux. Le but étant d'agrémenter le joueur humain, des idées sont proposées pour influencer la prise de décisions en ce sens. Au lieu d'optimiser les décisions pour gagner, des stratégies sont présentées pour maximiser l'expérience de jeu de l'adversaire.

Le chapitre 4 porte sur la planification de mouvements pour le robot AZIMUT-3. Il s'agit d'une contribution complémentaire aux trois premiers chapitres qui portent sur la planification d'actions concurrentes sous incertitude. En fait, dans les architectures robotiques, la planification des actions et des déplacements se fait généralement à l'aide de deux planificateurs distincts. Cependant, ces planificateurs collaborent : le premier décide de l'endroit où aller, alors que le second décide du chemin ou de la trajectoire à emprunter. Comme indiqué plus haut, pour planifier, le planificateur d'actions a besoin de simuler les conséquences des actions. Le robot AZIMUT-3, étant omnidirectionnel, peut se déplacer efficacement dans toutes les directions. Cela

présente un avantage considérable. Or, pour générer un plan d'actions, le premier planificateur a besoin d'estimer la durée des déplacements. Une façon d'estimer ces déplacements est de planifier des trajectoires.

La thèse se termine par une conclusion qui rappelle les objectifs et les principales contributions présentés. Des travaux futurs sont également proposés pour améliorer l'applicabilité des algorithmes de planification présentés. Ces limites sont inhérentes aux hypothèses simplificatrices requises par ces algorithmes.

Chapitre 1

Planification d'actions concurrentes avec des contraintes et de l'incertitude sur le temps et les ressources

Résumé

Les problèmes de planification combinant des actions concurrentes (simultanées) en plus de contraintes et d'incertitude sur le temps et les ressources représentent une classe de problèmes très difficiles en intelligence artificielle. Les méthodes existantes qui sont basées sur les processus décisionnels markoviens (MDP) doivent utiliser un modèle discret pour la représentation du temps et des ressources. Cette discrétisation est problématique puisqu'elle provoque une explosion exponentielle de l'espace d'états ainsi qu'un grand nombre de transitions. Ce chapitre présente ActuPlan, un planificateur basé sur une nouvelle approche de planification qui utilise un modèle continu plutôt que discret afin d'éviter l'explosion de l'espace d'états causée par la discrétisation. L'incertitude sur le temps et les ressources est représentée à l'aide de variables aléatoires continues qui sont organisées dans un réseau bayésien construit dynamique-

ment. Une représentation d'états augmentés associe ces variables aléatoires aux variables d'état. Deux versions du planificateur ActuPlan sont présentées. La première, nommée ActuPlan^{nc}, génère des plans non conditionnels à l'aide d'un algorithme de recherche à chainage avant dans un espace d'états augmentés. Les plans non conditionnels générés sont optimaux à une erreur près. Les plans générés satisfont à un ensemble de contraintes dures telles que des contraintes temporelles sur les buts et un seuil fixé sur la probabilité de succès des plans. Le planificateur ActuPlan^{nc} est ensuite adapté afin de générer un ensemble de plans non conditionnels qui sont caractérisés par différents compromis entre leur cout et leur probabilité de succès. Ces plans non conditionnels sont fusionnés par le planificateur ActuPlan afin de construire un meilleur plan conditionnel qui retarde des décisions au moment de l'exécution. Les branchements conditionnels de ces temps sont réalisés en conditionnant le temps. Des résultats empiriques sur des bancs d'essai classiques démontrent l'efficacité de l'approche.

Commentaires

L'article présenté dans ce chapitre sera soumis au *Journal of Artificial Intelligence Research (JAIR)*. Il représente la principale contribution de cette thèse. Cet article approfondit deux articles précédemment publiés et présentés à la vingtième *International Conference on Automated Planning and Scheduling* (ICAPS-2010) [9] et à la dix-neuvième *European Conference on Artificial Intelligence* (ECAI-2010) [10]. L'article présenté à ICAPS-2010 présente les fondements de base derrière la nouvelle approche de planification proposée, c'est-à-dire la combinaison d'un planificateur à chainage avant avec un réseau bayésien pour la représentation de l'incertitude sur le temps [9]. L'article présenté à ECAI-2010 généralise cette approche aux ressources continues [10]. En plus de détailler ces deux articles, l'article présenté dans ce chapitre va plus loin. Une extention est faite à la planification conditionnel. Des plans conditionnels sont construits en fusionnant des plans générés par le planificateur non conditionnel. L'article a été rédigé par Éric Beaudry sous la supervision de Froduald Kabanza et de François Michaud.

Planning with Concurrency under Time and Resource Constraints and Uncertainty

Éric Beaudry, Froduald Kabanza
Département d'informatique, Université de Sherbrooke,
Sherbrooke, Québec, Canada J1K 2R1
eric.beaudry@usherbrooke.ca, froduald.kabanza@usherbrooke.ca
François Michaud
Département de génie électrique et de génie informatique,
Université de Sherbrooke,
Sherbrooke, Québec, Canada J1K 2R1
francois.michaud@usherbrooke.ca

Abstract

Planning with action concurrency under time and resource constraints and uncertainty represents a challenging class of planning problems in AI. Current probabilistic planning approaches relying on a discrete model of time and resources are limited by a blow-up of the search state-space and of the number of state transitions. This paper presents ActuPlan, a planner based on a new planning approach which uses a continuous model of time and resources. The uncertainty on time and resources is represented by continuous random variables which are organized in a dynamically generated Bayesian network. A first version of the planner, ActuPlan^{nc}, performs a forward-search in an augmented state-space to generate near optimal nonconditional plans which are robust to uncertainty (threshold on the probability of success). ActuPlan^{nc} is then adapted to generate a set of nonconditional plans which are characterized by different trade-offs between their probability of success and their expected cost. A second version, ActuPlan, builds a conditional plan with a lower expected cost by merging previously generated nonconditional plans. The branches are built by conditioning on the time. Empirical experimentation on standard benchmarks demonstrates the effectiveness of the approach.

1.1 Introduction

Planning under time and resource constraints and uncertainty becomes a problem with a high computational complexity when the execution of concurrent (simultaneous) actions is allowed. This particularly challenging problem is motivated by real-world applications. One such application is the planning of daily activities for the Mars rovers [17]. Since the surface of Mars is only partially known and locally uncertain, the duration of navigation tasks is usually unpredictable. The amount of energy consumed by effectors and produced by solar panels is also subject to uncertainty. The generation of optimal plans for Mars rovers thus requires the consideration of uncertainty at planning time.

Another application involving both concurrency and uncertainty is the task planning for robots interacting with humans. From our experience at the AAAI Robot Challenge 2005 and 2006 [49], one difficulty we faced was the unpredictable duration of the human-robot interaction actions. Navigating in a crowd is difficult and makes the navigation velocity unstable. To address uncertainty, we adopted several non-optimal strategies like reactive planning [7] and we used a conservative planning model for the duration of actions. A more appropriate approach would require to directly consider uncertainty during planning. There exist other applications such as transport logistics which also have to deal with simultaneous actions and uncertainty on time and resources [4, 21].

The class of planning problems involving both concurrency and uncertainty is also known as *Concurrent Probabilistic Temporal Planning* (CPTP) [47]. Most state-of-the-art approaches handling this class of problems are based on Markov Decision Processes (MDPs), which is not surprising since MDPs are a natural framework for decision-making under uncertainty. A CPTP problem can be translated into a Concurrent MDP by using an interwoven state-space. Several methods have been developed to generate optimal and sub-optimal policies [45, 43, 46, 47, 56].

An important assumption required by most of these approaches is the time alignment of decision epochs in order to have a finite interwoven state-space. As these approaches use a discrete time and resources model, they are characterized by a huge state explosion. More specifically, in the search process, an action having an uncertain

duration produces several successor states each associated to different timestamps. Depending on the discretization granularity, this results in a considerable number of states and transitions. Thus, the scalability to larger problems is seriously limited by memory size and available time.

Simulation-based planning approaches represent an interesting alternative to produce sub-optimal plans. An example of this approach is the Generate, Test and Debug (GTD) paradigm [64] which is based on the integration of a deterministic planner and a plan simulator. It generates an initial plan without taking uncertainty into account, which is then simulated using a probabilistic model to identify potential failure points. The plan is incrementally improved by successively adding contingencies to the generated plan to address uncertainty. However, this method does not provide guaranties about optimality and completeness [46]. The Factored Policy Gradient (FPG) [18] is another planning approach based on policy-gradient methods borrowed from reinforcement learning [61]. However, the scalability of FPG is also limited and it does not generate optimal plans.

Because current approaches are limited in scalability or are not optimal, there is a need for a new and better approach. This paper presents ActuPlan, a planner based on a different planning approach to address both concurrency and numerical uncertainty [1]. Instead of using an MDP, the presented approach uses a deterministic forward-search planner combined with a Bayesian network. Two versions of ActuPlan are presented. The first planner one is ActuPlan^{nc} and generates non-conditional plans which are robust to numerical uncertainty, while using a continuous time model. More precisely, the uncertainty on the occurrences of events (the start and end time of actions) is modelled using continuous random variables, which are named *time random variables* in the rest of this paper. The probabilistic conditional dependencies between these variables are captured in a dynamically-generated Bayesian network. The state of resources (e.g., amount of energy or fuel) are also modelled by continuous random variables and are named *numerical random variables* and also added to the Bayesian network.

With this representation, ActuPlan^{nc} performs a forward-search in an aug-

1. In the rest of this paper, the expression *numerical uncertainty* is used to simply designate uncertainty on both time and continuous resources.

mented state-space where state features are associated to random variables to mark their valid time or their belief. During the search, the Bayesian network is dynamically generated and the distributions of random variables are incrementally estimated. The probability of success and the expected cost of candidate plans are estimated by querying the Bayesian network. Generated nonconditional plans are near optimal in terms of expected cost (e.g., makespan) and have a probability of success greater than or equal to a given threshold. These plans are well suited for agents which are constrained to execute deterministic plans.

ActuPlan, the second planner, merges several nonconditional plans. The nonconditional planning algorithm is adapted to generate several nonconditional plans which are characterized by different trade-offs between their probability of success and their expected cost. The resulting conditional plan has a lower cost and/or a higher probability of success than individual nonconditional plans. At execution time, the current observed time is tested in order to select the best execution path in the conditional plan. The switch conditions within the plan are built through an analysis of the estimated distribution probability of random variables. These plans are better suited for agents allowed to make decisions during execution. In fact, conditional plans are required to guarantee the optimal behaviour of agents under uncertainty because many decisions must be postponed to execution time.

This paper extends two previously published papers about planning with concurrency under uncertainty on the duration of actions and on resources. The first one presented the basic concepts of our approach, i.e. the combination of a deterministic planner with a Bayesian network [9]. The second one generalized this approach to also consider uncertainty on continuous resources [10]. This paper goes further and presents a conditional planner generating plans which postpone some decisions to execution time in order to lower their cost.

The rest of this paper is organized as follows. Section 1.2 introduces the definition of important structures about states, actions, goals and plans. Section 1.3 introduces the nonconditional planning algorithm (ActuPlan^{nc}) and related concepts. Section 1.4 presents the conditional planning approach (ActuPlan). Experiments are reported in Section 1.5. Section 1.6 discusses related works. We conclude with a summary of the contributions and ideas for future work.

1.2 Basic Concepts

The concepts presented herein are illustrated with examples based on the Transport domain taken from the International Planning Competition [2] of the International Conference on Automated Planning and Scheduling. In that planning domain, which is simplified as much as possible to focus on important concepts, trucks have to deliver packages. A package is either at a location or loaded onto a truck. There is no limit on the number of packages a truck can transport at the same time and on the number of trucks that can be parked at same location.

1.2.1 State Variables

State features are represented as state variables [53]. There are two types of *state variables*: object variables and numerical variables. An *object state variable* $x \in X$ describes a particular feature of the world state associated to a finite domain $Dom(x)$. For instance, the location of a robot can be represented by an object variable whose domain is the set of all locations distributed over a map.

A *numerical state variable* $y \in Y$ describes a numerical feature of the world state. A resource like the current energy level of a robot's battery is an example of a state numerical variable. Each numerical variable y has a valid domain of values $Dom(y) = [y_{min}, y_{max}]$ where $(y_{min}, y_{max}) \in \mathbb{R}^2$. The set of all state variables is noted $Z = X \cup Y$. A world state is an assignment of values to the state variables, while action effects (state updates) are changes of variable values. It is assumed that no exogenous events take place; hence only planned actions cause state changes.

For instance, a planning problem in the Transport domain has the following objects: a set of n trucks $R = \{r_1, \ldots, r_n\}$, a set m of packages $B = \{b_1, \ldots, b_m\}$ and a set of k locations $L = \{l_1, \ldots, l_k\}$ distributed over a map. The set of object state variables $X = \{C_r, C_b \mid r \in R, b \in B\}$ specifies the current location of trucks and packages. The domain of object variables is defined as $Dom(C_r) = L$ $(\forall r \in R)$ and $Dom(C_b) = L \cup R$ $(\forall b \in B)$. The set of numerical state variables $Y = \{F_r \mid r \in R\}$ specifies the current fuel level of trucks.

2. http://planning.cis.strath.ac.uk/competition/domains.html

1.2.2 Time and Numerical Random Variables

The uncertainty related to time and numerical resources is represented using continuous random variables. A *time random variable* $t \in T$ marks the occurrence of an event, corresponding to either the start or the end of an action. An event induces a change of the values of a subset of state variables. The time random variable $t_0 \in T$ is reserved for the initial time, i.e., the time associated to all state variables in the initial world state. Each action a has a duration represented by a random variable d_a. A time random variable $t \in T$ is defined by an equation specifying the time at which the associated event occurs. For instance, an action a starting at time t_0 will end at time t_1, the latter being defined by the equation $t_1 = t_0 + d_a$.

Uncertainty on the values of numerical state variables is also modelled by random variables. Instead of crisp values, state variables have as values *numerical random variables*. We note N the set of all numerical random variables. A numerical random variable is defined by an equation which specifies its relationship with other random variables. For instance, let y be a numerical state variable representing a particular resource. The corresponding value would be represented by a corresponding random variable, let's say $n_0 \in N$. Let the random variable $cons_{a,y}$ represent the amount of resource y consumed by action a. The execution of action a changes the current value of y to a new random variable n_1 defined by the equation $n_1 = n_0 - cons_{a,y}$.

1.2.3 States

A state describes the current world state using a set of state features, that is, a set of value assignations for all state variables.

A *state* s is defined by $s = (\mathcal{U}, \mathcal{V}, \mathcal{R}, \mathcal{W})$ where:

- $\mathcal{U}$ is a total mapping function $\mathcal{U} : X \to \cup_{x \in X} Dom(x)$ which retrieves the *current assigned value* for each object variable $x \in X$ such that $\mathcal{U}(x) \in Dom(x)$.
- $\mathcal{V}$ is a total mapping function $\mathcal{V} : X \to T$ which denotes the *valid time* at which the assignation of variables X have become effective.
- $\mathcal{R}$ is a total mapping function $\mathcal{R} : X \to T$ which indicates the *release time* on object state variables X which correspond to the latest time that *over all* conditions expire.

– $\mathcal{W}$ is a total mapping function $\mathcal{W} : Y \rightarrow N$ which denotes the *current belief* of numerical variables Y.

The release times of object state variables are used to track *over all* conditions of actions. The time random variable $t = \mathcal{R}(x)$ means that a change of the object state variable x cannot be initiated before time t. The valid time of an object state variable is always before or equals to its release time, i.e., $\mathcal{V}(x) \leq \mathcal{R}(x) \forall x \in X$.

The valid time ($\mathcal{V}$) and the release time ($\mathcal{R}$) respectively correspond to the write-time and the read-time in Multiple-Timeline of SHOP2 planner [28], with the key difference here being that random variables are used instead of numerical values.

Hence, a state is not associated with a fixed timestamp as in a traditional approach for action concurrency [2]. Only numerical uncertainty is considered in this paper, i.e., there is no uncertainty about the values being assigned to object state variables. Uncertainty on object state variables is not handled because with do not address actions with uncertainty on their outcomes. Dealing with this kind of uncertainty is planned as future work. The only uncertainty on object state variables is about **when** their assigned values become valid. The valid time $\mathcal{V}(x)$ models this uncertainty by mapping each object state variable to a corresponding time random variable.

Figure 1.1 illustrates an example of a state in the Transport domain. The left side (a) is illustrated using a graphical representation based on a topological map. The right side (b) presents state s_0 in ActuPlan formalism. Two trucks r_1 and r_2 are respectively located at locations l_1 and l_4. Package b_1 is loaded on r_1 and package b_2 is located at location l_3. The valid time of all state variables is set to time t_0.

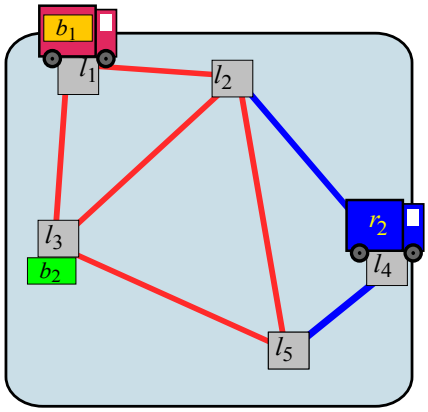
1.2.4 Actions

The specification of actions follows the extensions introduced in PDDL 2.1 [31] for expressing temporal planning domains. The set of all actions is denoted by A. An *action* $a \in A$ is a tuple a=($name$, $cstart$, $coverall$, $estart$, $eend$, $enum$, d_a) where :

– *name* is the name of the action;

– *cstart* is the set of *at start* conditions that must be satisfied at the beginning of the action;

– *coverall* is the set of persistence conditions that must be satisfied *over all* the



(a) Graphical representation

s_0			
x	$\mathcal{U}(x)$	$\mathcal{V}(x)$	$\mathcal{R}(x)$
C_{r1}	l_1	t_0	t_0
C_{r2}	l_4	t_0	t_0
C_{b1}	r_1	t_0	t_0
C_{b2}	l_3	t_0	t_0
y	$\mathcal{W}(y)$		
F_{r1}	n_0		
F_{r2}	n_0		

(b) State representation

Figure 1.1: Example of a state for the Transport domain

duration of the action;

- *estart* and *eend* are respectively the sets of *at start* and *at end* effects on the state object variables;
- *enum* is the set of numerical effects on state numerical variables;
- and $d_a \in D$ is the random variable which models the duration of the action.

A condition c is a Boolean expression over state variables. The function $vars(c) \rightarrow 2^X$ returns the set of all object state variables that are referenced by the condition c.

An object effect $e = (x, exp)$ specifies the assignation of the value resulting from the evaluation of expression exp to the object state variable x. The expressions $conds(a)$ and $effects(a)$ return, respectively, all conditions and all effects of action a.

A numerical effect is either a change e_c or an assignation e_a. A numerical change $e_c = (y, numchange_{a,y})$ specifies that the action changes (increases or decreases) the numerical state variable y by the random variable $numchange_{a,y}$. A numerical assignation $e_a = (y, newvar_{a,y})$ specifies that the numerical state variable y is set to the random variable $newvar_{a,y}$.

The set of *action duration random variables* is defined by $D = \{d_a \mid a \in A\}$ where A is the set of actions. A random variable d_a for an action follows a probability distribution specified by a probability density function $\phi_{d_a} : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ and a cumulative distribution function $\Phi_{d_a} : \mathbb{R}^+ \rightarrow [0, 1]$.

An action a is **applicable** in a state s if all the following conditions are satisfied:

1. state s satisfies all **at start** and **over all** conditions of a. A condition $c \in conds(a)$ is satisfied in state s if c is satisfied by the current assigned values of state variables of s.
2. All state numerical variables $y \in Y$ are in a valid state, i.e., $W(y) \in [y_{min}, y_{max}]$.

Since the value of a numerical state variable is probabilistic, its validity is also probabilistic. The application of an action may thus cause a numerical state variable to become invalid. We denote $P(\mathcal{W}(y) \in Dom(y))$ the probability that a numerical state variable y be in a valid state when its belief is modelled by a numerical random variable $\mathcal{W}(y) \in N$.

Table 1.1 presents actions for the Transport domain .

Table 1.1: Actions specification of the Transport domain

$Goto(r, l_a, l_b)$	
$cstart$	$C_r = l_a$
$eend$	$C_r = l_b$
$duration$	$Normal(dist/speed, 0.2 * dist/speed)$
$enum$	F_r-=$Normal(dist/rate, 0.3 * dist/rate)$
$Load(r, l, b)$	
$cstart$	$C_b = l$
$coverall$	$C_r = l$
$eend$	$C_b = r$
$duration$	$Uniform(30, 60)$
$Unload(r, l, b)$	
$cstart$	$C_b = r$
$coverall$	$C_r = l$
$eend$	$C_b = l$
$duration$	$Uniform(30, 60)$
$Refuel(r, l)$	
$coverall$	$C_r = l$
$enum$	$F_r = F_{max,r}$
$duration$	$Uniform(30, 60)$

1.2.5 Dependencies on Action Duration Random Variables

Bayesian networks provide a rich framework to model complex probabilistic dependencies between random variables [25]. Consequently, the use of continuous random variables organized in a Bayesian network provides a flexibility for modelling probabilistic dependencies between the durations of actions. Few assumptions about the independence or the dependence of durations are discussed in this section.

The simplest case is to make the assumption that all actions have independent durations. Under the independence assumption, the duration of two arbitrary actions a and b can be modelled by two independent random variables d_a and d_b. However, this assumption may be not realistic for planning applications having actions with dependent durations. Let actions a and b represent the move of two trucks in traffic. If it is known that one truck is just following the other, it is reasonable to say that both actions should have approximately the same duration. This claim is possible because the uncertainty is not directly related to actions but to the path. This situation can be easily modelled in a Bayesian network by inserting an additional random variable d_{path} which represents the duration of taking a particular path. Consequently, random variables d_a and d_b directly depend on d_{path}.

Another important consideration concerns several executions of the same action. Let action a represent the action of moving a truck on an unknown path. Since the path is unknown, the duration of moving action is then probabilistic. Once the path is travelled for the first time, it may be reasonable to say that future travels along the same path will take approximately the same time. Hence we consider that if the duration of an execution of a is modelled by random variable d_a which follows the normal distribution $\mathcal{N}(\mu, \sigma)$ [3] , executing action a twice has the total duration $2d_a$ which follows $\mathcal{N}(2\mu, 2\sigma)$. It may also be the case that all executions of a same action have independent durations. For instance, the uncertainty about a may come from the traffic which is continuously changing. This can be modelled using one random variable per execution. Thus the total duration of two executions of a corresponds to $d_{a,1} + d_{a,2}$ which follows $\mathcal{N}(2\mu, \sqrt{2}\sigma)$.

This discussion about dependence or independence assumptions on action dura-

3. In this thesis, the notation $\mathcal{N}(\mu, \sigma)$ is used instead of $\mathcal{N}(\mu, \sigma^2)$.

tion random variables can be generalized to the uncertainty on numerical resources. Depending on the planning domain, it is also possible to create random variables and dependencies between time and numerical random variables to model more precisely the relationship between the consumption and the production of resources and the duration of actions.

1.2.6 State Transition

Algorithm 1 Apply action function

```
1.  function Apply(s, a)
2.     s' ← s
3.     t_conds ← max_{x∈vars(conds(a))} s.V(x)
4.     t_release ← max_{x∈vars(effects(a))} s.R(x)
5.     t_start ← max(t_conds, t_release)
6.     t_end ← t_start + d_a
7.     for each c ∈ a.coverall
8.        for each x ∈ vars(c)
9.           s'.R(x) ← max(s'.R(x), t_end)
10.    for each e ∈ a.estart
11.       s'.U(e.x) ← eval(e.exp)
12.       s'.V(e.x) ← t_start
13.       s'.R(e.x) ← t_start
14.    for each e ∈ a.eend
15.       s'.U(e.x) ← eval(e.exp)
16.       s'.V(e.x) ← t_end
17.       s'.R(e.x) ← t_end
18.    for each e ∈ a.enum
19.       s'.W(e.y) ← eval(e.exp)
20.    returns s'
```

The planning algorithm expands a search graph in the state space and dynamically generates a Bayesian network which contains random variables.

Algorithm 1 describes the Apply function which computes the state resulting from application of an action a to a state s. Time random variables are added to the Bayesian network when new states are generated. The start time of an action is defined as the earliest time at which its requirements are satisfied in the current state. Line 3 calculates the time t_{conds} which is the earliest time at which all *at start* and *over all* conditions are satisfied. This time corresponds to the maximum of all time random

variables associated to the state variables referenced in the action's conditions. Line 4 calculates time $t_{release}$ which is the earliest time at which all persistence conditions are released on all state variables modified by an effect. Then at Line 5, the time random variable t_{start} is generated. Its defining equation is the max of all time random variables collected in Lines 3–4. Line 6 generates the time random variable t_{end} with the equation $t_{end} = t_{start} + d_a$. Once generated, the time random variables t_{start} and t_{end} are added to the Bayesian network if they do not already exist. Lines 7–9 set the release time to t_{end} for each state variable involved in an *over all* condition. Lines 10–17 process *at start* and *at end* effects. For each effect on a state variable, they assign this state variable a new value, set the valid and release times to t_{start} and add t_{end}. Line 18–19 process numerical effects.

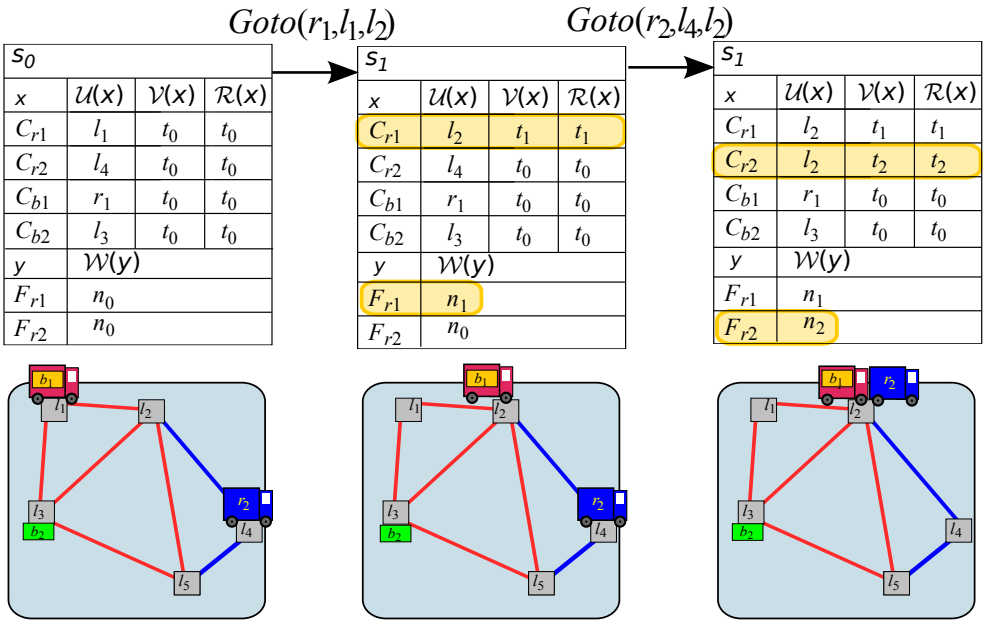


Figure 1.2: Example of two state transitions in the Transport domain

Figure 1.2 illustrates two state transitions. State s_1 is obtained by applying the action $Goto(r_1, l_1, l_2)$ from state s_0. The Apply function (see Algorithm 1) works as follows. The action $Goto(r_1, l_1, l_2)$ has the *at start* condition $C_{r_1} = l_1$. Because C_{r_1} is associated to t_0, we have $t_{conds} = \max(t_0) = t_0$. Since the action modifies the C_{r_1} object state variable, Line 4 computes the time $t_{release} = \max(t_0) = t_0$. At

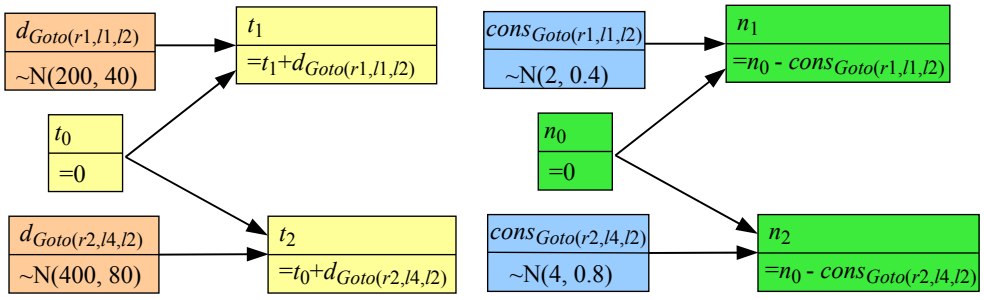


Figure 1.3: Example of a Bayesian network expanded after two state transitions

Line 5, the action's start time is defined as $t_{start} = \max(t_{conds}, t_{release}) = t_0$, which already exists. Then, at Line 6, the time random variable $t_{end} = t_0 + d_{Goto(r_1,l_1,l_2)}$ is created and added to the Bayesian network with the label t_1. Figure 1.3 presents the corresponding Bayesian network. Next, Lines 13–16 apply effects by performing the assignation $C_{r_1} = l_2$ and by setting time t_1 as the valid time for C_{r_1}. The application of the numerical effect creates a new numerical random variable n_1 which is associated to the belief of F_{r_1} in state s_1. As shown in the Bayesian network, n_1 is defined by $n_1 = n_0 - cons_{Goto(r_1,l_1,l_2)}$ where $cons_{Goto(r_1,l_1,l_2)}$ is a random variable representing the fuel consumption by the action. State s_2 is obtained similarly by applying the action $Goto(r_2, l_4, l_2)$ from state s_1. Since both actions start at time t_0, they are started simultaneously.

1.2.7 Goals

A *goal* $\mathcal{G}$ is a conjunction of deadline conditions over state features. A deadline condition is a tuple $(x, v, dl) \in \mathcal{G}$ meaning that state variable $x \in X$ has to be assigned the value $v \in Dom(x)$ within deadline $dl \in \mathbb{R}^+$ time. In this paper, a goal is noted by the syntax $\mathcal{G} = \{x_1 = v_1@dl_1, \ldots, x_n = v_n@dl_n\}$. When a goal condition has no deadline ($dl = +\infty$), we simply write $x = v$, i.e. $@dl = +\infty$ is optional.

For a given state and goal, $s \models G$ denotes that all conditions in G are satisfied in s. Because the time is uncertain, the satisfaction of a goal in a state ($s \models G$) is implicitly a Boolean random variable. Thus, $P(s \models \mathcal{G})$ denotes the probability that state s satisfies goal $\mathcal{G}$.

Note that if a goal has deadlines, then $P(s \models \mathcal{G}) < 1$ generally holds because

actions may have a non-zero probability of infinite duration. For that reason, a goal is generally associated with a threshold α on the desired probability of success. Consequently, a planning problem is defined by $(s_0, \mathcal{G}, \alpha)$ where s_0 is the initial state.

1.2.8 Plans

A plan is a structured set of actions which can be executed by an agent. Two types of plans are distinguished in this paper. A *nonconditional plan* is a plan that cannot change the behaviour of the agent depending on its observations (e.g. the actual duration of actions). Contrary to a nonconditional plan, a *conditional plan* takes advantage of observations during execution. Hence, the behaviour of an agent can change according to the actual duration of actions. Generally, a conditional plan enables a better behaviour of an agent because it provides alternative ways to achieve its goals.

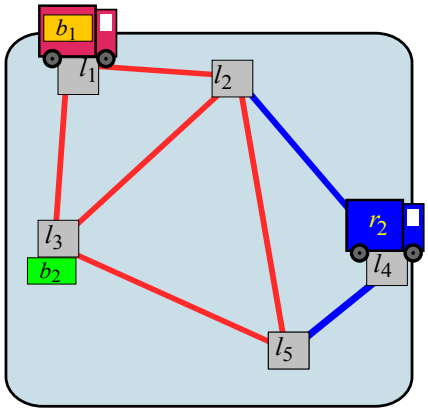
A **nonconditional plan** π is defined by a partially ordered set of actions $\pi = (A_\pi, \prec_\pi)$ where:

- A_π is a set of labelled actions noted $\{label_1{:}a_1, \ldots, label_n{:}a_n\}$ with $a_i \in A$; and
- $\prec_\pi$ is a set of precedence constraints, each one noted $label_i \prec \ldots \prec label_j$.

The definition of a plan requires labelled actions because an action can be executed more than one time in a plan. During execution, actions are executed as soon as their precedence constraints are satisfied.

Let s_0 be the initial state in Figure 1.4 and $\mathcal{G} = \{(C_{b_1} = l_4)\}$ be a goal to satisfy. The plan $\pi = (\{a_1{:}\ \mathit{Goto}(r_1, l_1, l_2),\ a_2{:}\ \mathit{Unload}(r_1, l_2, b_1),\ a_3{:}\ \mathit{Goto}(r_2, l_4, l_2),\ a_4{:}\ \mathit{Load}(r_2, l_2, b_1),\ a_5{:}\ \mathit{Goto}(r_2, l_2, l_4),\ a_6{:}\ \mathit{Unload}(r_2, l_4, b_1)\},\ \{a_1 \prec a_2 \prec a_4,\ a_3 \prec a_4 \prec a_5 \prec a_6\ \})$ is a possible solution plan to the problem. This plan starts two independent actions a_1 and a_3. Actions a_2 is started as soon as a_1 is finished. Once both a_2 and a_3 are finished, a_4 is started. Finally, a_5 and a_6 are executed sequentially after a_4.

A **conditional plan** π is defined as a finite state machine (FSM), where each state contains time switching conditions. In each state, an action is selected to be started depending on the current time which depends on how long the execution of previous actions have taken. A example of conditional plan and its construction are



(a) Graphical representation

s_0			
x	$\mathcal{U}(x)$	$\mathcal{V}(x)$	$\mathcal{R}(x)$
C_{r1}	l_1	t_0	t_0
C_{r2}	l_4	t_0	t_0
C_{b1}	r_1	t_0	t_0
C_{b2}	l_3	t_0	t_0
y	$\mathcal{W}(y)$		
F_{r1}	n_0		
F_{r2}	n_0		

(b) State representation

Figure 1.4: An initial state in the Transport domain

presented in Section 1.4.

The notation $\pi \models \mathcal{G}$ is used to denote that the plan π is a solution to reach a state which satisfies the goal $\mathcal{G}$.

1.2.9 Metrics

The quality of a plan π is evaluated by a given metric function $cost(\pi)$. This evaluation is made from an implicit initial state s_0. Typical cost functions are :

- the expected makespan denoted $E[makespan(\pi)]$;
- the sum of the cost of actions;
- a formula evaluated on the last state s reached by the exection of π;
- or a linear combination of the expected makespan, the sum of the cost of actions and of a formula.

In this paper, makespan is used as the cost function for examples. The makespan of a plan π is computed from the last state s which is reached by its execution and is evaluated by Equation (1.1). Note that the makespan of a plan is a random variable.

$$makespan(\pi) = \max_{x \in X} s.\mathcal{V}(x) \tag{1.1}$$

1.3 ActuPlan^{nc} : Nonconditional Planner

ActuPlan^{nc} is the nonconditional planner version of ActuPlan. It performs a forward-search in the state space. ActuPlan handles actions' delayed effects, i.e. an effect specified to occur at a given point in time after the start of an action. The way the planner handles concurrency and delayed effects is slightly different from a traditional concurrency model such as the one used in TLPlan [2]. In this traditional model, a state is augmented with a timestamp and an event-queue (agenda) which contains delayed effects. A special *advance-time* action triggers the delayed effects whenever appropriate.

In ActuPlan, the time is not directly attached to states. As said in Section 1.2.3, ActuPlan adopts a strategy similar to Multiple-Timeline as in SHOP2 [52]. Time is not attached to states, it is rather associated with state features to mark their valid time. However, contrary to SHOP2, our planner manipulates continuous random variables instead of numerical timestamps. As a consequence, the planner does not need to manage an events-queue for delayed effects and the special advance-time action. The time increment is tracked by the time random variables attached to time features. A time random variable for a feature is updated by the application of an action only if the effect of the action changes the feature; the update reflects the delayed effect on the feature.

Algorithm 2 presents the planning algorithm of ActuPlan^{nc} in a recursive form. This planning algorithm performs best-first-search [4] in the state space to find a state which satisfies the goal with a probability of success higher than a given threshold α (Line 2). The α parameter is a constraint defined by the user and is set according to his fault tolerance. If $s \models \mathcal{G}$ then a nonconditional plan π is built (Line 3) and returned (Line 4). The choice of an action a at Line 5 is a backtrack point. A heuristic function is involved to guide this choice (see Section 1.3.3). The optimization criteria is implicitly given by a given metric function *cost* (see Section 1.2.9). Line 6 applies the chosen action to the current state. At Line 7, an upper bound on the probability that state s can lead to a state which satisfies goal $\mathcal{G}$ is evaluated. The symbol $\models^*$ is used to denote a goal may be reachable from a given state, i.e. their may exist a

4. To be not confused with Greedy best-first-search [57].

plan. The symbol $\overline{P}$ denotes an upper bound on the probability of success. If that probability is under the fixed threshold α, the state s is pruned. Line 8 performs a recursive call.

Algorithm 2 Nonconditional planning algorithm

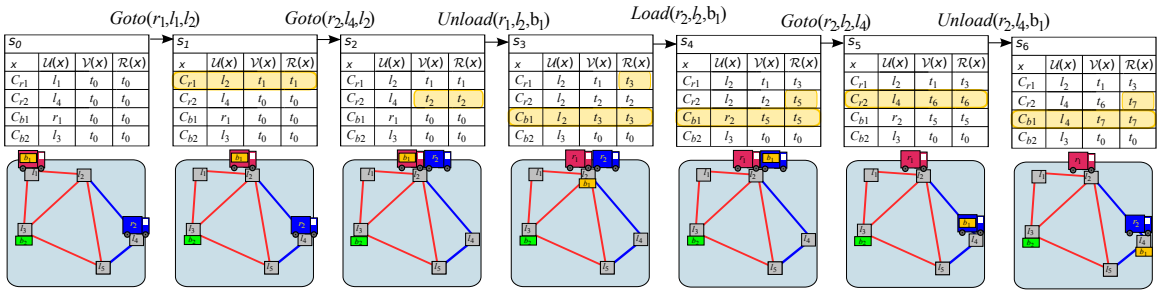
```
1.  ActuPlan^nc(s, G, A, α)
2.    if P(s ⊨ G) ≥ α
3.      π ← ExtractNonConditionalPlan(s)
4.      return π
5.    nondeterministically choose a ∈ A
6.    s' ← Apply(s, a)
7.    if P̄(s' ⊨* G) ≥ α
8.      return ActuPlan^nc(s', G, A, α)
9.    else return FAILURE
```

1.3.1 Example on Transport domain

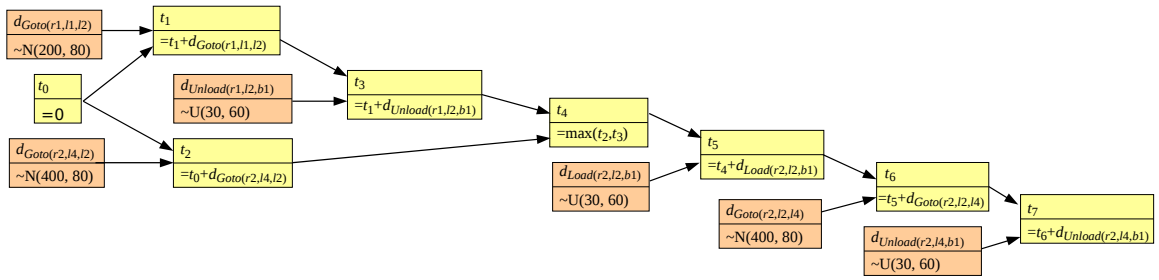
Figure 1.5 illustrates an example of a partial search carried out by Algorithm 2 on a problem instance of the Transport domain. The goal in that example is defined by $\mathcal{G} = \{C_{b_1} = l_4\}$. Note that trucks can only travel on paths of its color. For instance, truck r_1 cannot move from l_2 to l_4. A subset of expanded states is shown in (a). The states s_0, s_1 and s_2 are the same as previous figures except that numerical state variables, F_{r_1} and F_{r_2}, are not presented to save space.

State s_3 is obtained by applying $Unload(r_1, l_2, b_1)$ action from state s_2. This action has two conditions : the *at start* condition $C_{b_1} = r_1$ and the *over all* condition $C_{r_1} = l_2$. The action has the *at end* $C_{b_1} = l_2$ effect. The start time of this action is obtained by computing the maximum time of all valid times of state variables concerned by conditions and all release times of state variables concerned by effects. The start time is then $\max(t_0, t_1) = t_1$. The end time is $t_3 = t_1 + d_{Unload(r_1,l_2,b_1)}$. Because the *over all* condition $C_{r_1} = l_2$ exits, the release time $\mathcal{R}(C_{r_1})$ is updated to t_3. This means that another action cannot move truck r_1 away from l_2 before t_3.

State s_4 is obtained by applying $Load(r_2, l_2, b_1)$ action from state s_3. This action has two conditions : the *at start* condition $C_{b_1} = l_2$ and the *over all* condition $C_{r_2} = l_2$. The action has the *at end* $C_{b_1} = r_2$ effect. The start time is then $t_4 = \max(t_2, t_3)$.



(a) State-space



(b) Bayesian network

Figure 1.5: Sample search with the Transport domain

The end time is $t_5 = t_4 + d_{Load(r_2,l_2,b_1)}$. The goal $\mathcal{G}$ is finally satisfied in state s_6. Figure 1.6 shows the extracted nonconditional plan.

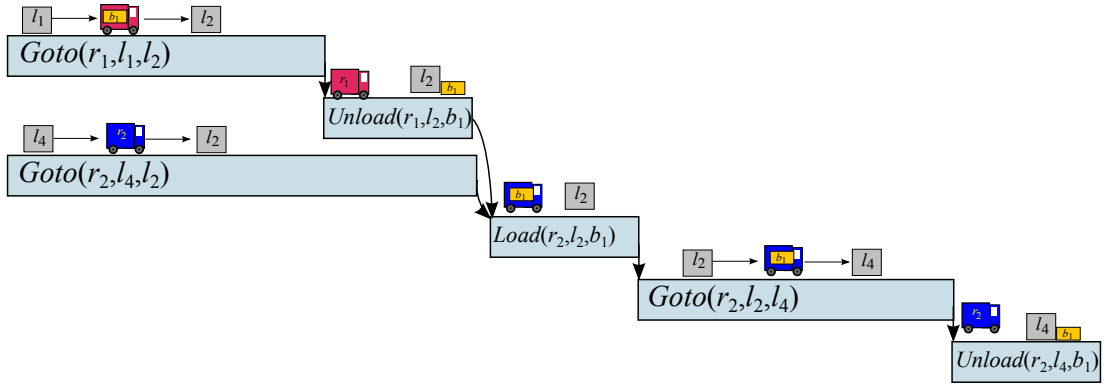


Figure 1.6: Extracted nonconditional plan

1.3.2 Bayesian Network Inference Algorithm

A Bayesian network inference algorithm is required to estimate the probability of success and the expected cost (e.g., makespan) of plans. The computation of these values requires the estimation of the distribution of the various random variables involved. In general, the choice of an inference algorithm for Bayesian networks is guided by the structure of the Bayesian network and by the type of random variables it includes [25]. In our case, the Bayesian network contains continuous random variables. Analytical inference methods are possible if some restrictions can be imposed on the variables probability distributions. In particular, normal distributions are often used because they can be defined by two parameters (mean μ and standard deviation σ), which makes them suitable for analytical approaches.

In our approach, the time random variables ($t \in T$) cannot be constrained to follow normal distributions because their equations may contain several instances of the *max* operator which appears in the Apply function. Even if two random variables t_1 and t_2 are normally distributed, the resulting random variable $t_3 = \max(t_1, t_2)$ is not normally distributed. Therefore, our approach leads to arbitrary forms of probabilistic distributions.

Because there exists no exact and analytical method for Bayesian networks having arbitrary types of distribution, approximate inference algorithms have to be used [25]. A direct sampling algorithm for the Bayesian network inferences is adopted [25]. This algorithm consists in simulating the whole Bayesian network. A topological sort is made on the random variables. Root random variables are initially sampled from their following distribution. Other random variables are then sampled as soon as their parents have been sampled. The samples of non-root random variables are generated by evaluating their equations, which involve the samples of their parents. Once all random variables are processed, exactly one sample has been generated for each random variable. By running n independent simulations, an array of n independent samples $m_1, \ldots, m_n$ are generated for each random variable. From an array of samples of a random variable, a sample mean $\hat{\mu}$ is evaluated by Equation (1.2).

$$\hat{\mu} = \frac{\sum_i^n m_i}{n} \tag{1.2}$$

Since a finite number of samples are generated for each random variable, $\hat{\mu}$ is an estimator of the real expected value $\mu = E[t]$. Some guaranties about the quality of the estimation can be given. This quality is described by an absolute error e under a given confidence level of γ. A confidence level γ means that $|\hat{\mu} - \mu| \leq e$ holds with a probability of γ. The error e can be estimated as follows. Each generated sample m_i is a random variable which follows the distribution of the estimated random variable. By the central limit theorem, the samples mean $\hat{\mu} = \frac{m_1}{n} + \cdots + \frac{m_n}{n}$ asymptotically follows a normal distribution $\mathcal{N}(\mu, \frac{\sigma}{\sqrt{n}})$ when the number of samples n increases. In practice, since the real standard deviation σ of X is generally unknown, an estimated $\hat{\sigma}$ can be used and is defined by Equation (1.3) [25, 40].

$$\hat{\sigma} = \sqrt{\frac{\sum_i^n (m_i - \hat{\mu})^2}{n-1}} \tag{1.3}$$

Consequently, the error $\hat{\mu} - \mu$ follows the normal distribution $\mathcal{N}(0, \frac{\hat{\sigma}}{\sqrt{n}})$. The probability that the estimated mean is under a fixed maximum error e is given by Equation (1.4) where $\Phi(\cdot)$ is the cumulative probability function of $\mathcal{N}(0, \frac{\hat{\sigma}}{\sqrt{n}})$. Equation (1.5) is then obtained.

$$\gamma = P(|\hat{\mu} - \mu| < e) = \Phi(e) - \Phi(-e) \tag{1.4}$$

$$\Phi(-e) = \frac{1-\gamma}{2} \tag{1.5}$$

If γ is a fixed confidence level, a maximum error e is estimated by Equation (1.6).

$$e = \Phi^{-1}\left(\frac{1-\gamma}{2}\right) \tag{1.6}$$

Equation (1.7) is then obtained by using the cumulative probability function of normal distribution $\mathcal{N}(1, 0)$. The absolute estimation error of the expected mean of random variables is proportional to the standard deviation σ of X and inversely proportional to the squared root of the number of samples n.

$$e = \Phi^{-1}_{\mathcal{N}(1,0)}\left(\frac{1-\gamma}{2}\right) \times \frac{\hat{\sigma}}{\sqrt{n}} \tag{1.7}$$

Incremental Estimation

The Bayesian network is constructed dynamically during the search process, more precisely when the Apply function (Algorithm 1) is invoked. Once a new numerical random variable or a new time random variable is created (see t_{start} and t_{end} in Algorithm 1), it is added to the Bayesian network and its probability distribution is immediately estimated. The distributions of random variables are required by the heuristic function to guide the planning search, and to estimate the probability that a candidate plan satisfies the goal. Because the Bayesian network is generated dynamically, we want to avoid evaluating the entire Bayesian network every time a new random variable is added. In the worst case, adopting this strategy would indeed require $n(n-1)/2$ evaluations for a network of n nodes.

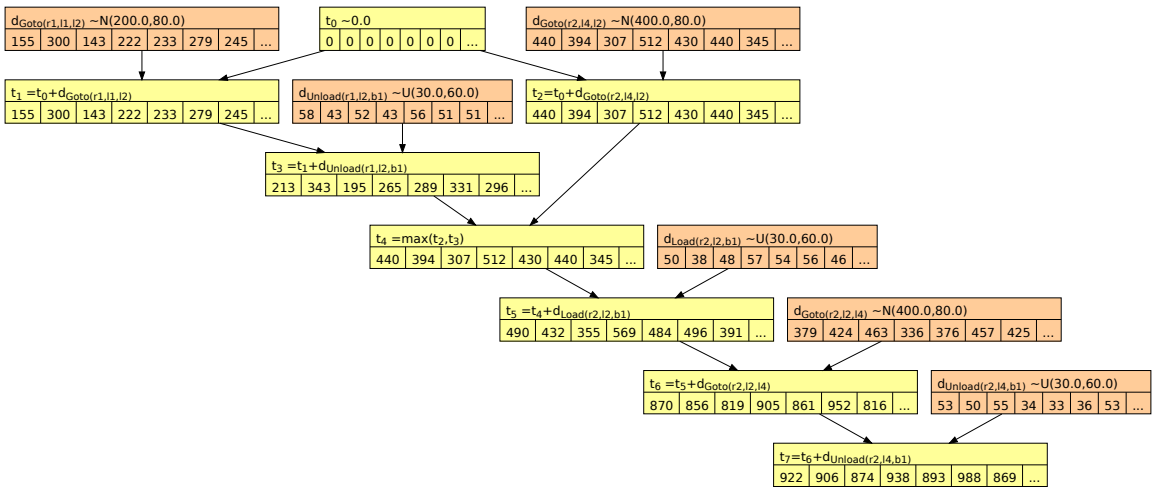


Figure 1.7: Random variables with samples

To reduce computation time, the generated samples are kept in memory. More precisely, each random variable t has an array of m samples $M_x = \langle m_{t,1}, m_{t,2}, ..., m_{t,m} \rangle$. Figure 1.7 shows a partial inference on the Bayesian network of the previous example (Figure 1.5 (b)). The i^{th} samples of all random variables correspond to an independent simulation of the entire Bayesian network. When a new random variable is added, it is evaluated by generating a new array of samples. The values of these samples are computed by using the samples of the parent variables, which are already available because they were kept in memory. This incremental strategy makes the computation

time equivalent to estimating the entire Bayesian network once.

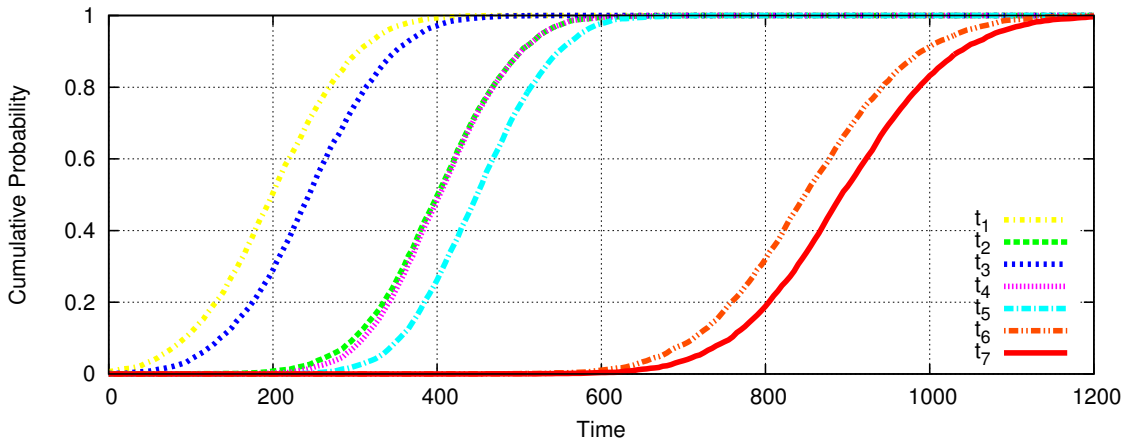


Figure 1.8: Estimated cumulative distribution functions (CDF) of random variables

Figure 1.8 shows the estimation of cumulative distribution functions of time random variables. The large red curve (t_7) represents the probability that package b_1 has arrived to location l_4 in function of a given time. For instance, if the deadline is time 1000 then the probability of satisfying it is greater than 0.8.

Samples Caching

The incremental belief estimation of random variables costs $O(nm)$ in memory size where n is the number of random variables and m is the number of samples. Since the number of samples can be relatively large (few thousands), this method is limited by the size of available memory in practice.

To cope this limitation, we propose a caching mechanism to only keep the generated samples of the most recently accessed random variables. During the search, a lot of states are added to the open and closed sets without being accessed until much later. The random variables attached to these states are also rarely accessed. Hence, the samples of these random variables may be temporarily freed. The freed samples could easily be regenerated on demand in the future. To guarantee the generation of exactly the same array of samples, root nodes in the Bayesian network are always kept in memory. Remember that root nodes correspond to the duration of actions and

the use of resources. The memory requirement for keeping those arrays of samples in memory is linearly proportional to the number of applicable grounded actions in the planning domain. The impact of the samples caching mechanism is evaluated in Section 1.5.2.

1.3.3 Minimum Final Cost Heuristic

Forward-search-based planners generally require heuristics to be efficient. Heuristics can be dependent or independent to the domain. Dependent heuristics are less general because they require to be specified by the user.

Basically, domain independent heuristics evaluate or estimate a minimum distance to the goal. For instance, HSP [15] and Fast-Forward (FF) [33] planners use a relaxed planning graph as a heuristic. This technique has been adapted in SAPA for temporal planning [28]. ActuPlan adapts this technique to time uncertainty.

Algorithm 3 presents a heuristic function which estimates a minimum final cost of a plan which passes by the state s. The returned value is slightly different from best-first-search and A^* where heuristics estimate h to compute $f = g + h$. The notion of remaining cost from a state to a goal is ill-defined in our approach because a state does not have a unique time. A state may have several state variables, each one associated to different time random variables. Therefore, the heuristic directly evaluates a lower bound on the total final cost (f in A^*) instead of estimating the remaining cost (h).

Algorithm 3 proceeds as follows. The minimal cost of each possible assignment of each state variable is tracked into the mapping function $C_{min} : (X, Dom(x)) \rightarrow \mathbb{R}$. Line 2 initializes the C_{min} function to $+\infty$ over its domain. Then at Lines 3–4, the minimum cost for the current value of each state variable is set to its valid time. Line 6 loops on all possible actions $a \in A$. If an action can reduce the minimum cost of an assignment $x = v$ then $C_{min}(x, v)$ is updated. The loop of Line 5 performs updates until there is no minimum cost reduction or until the goal is satisfied. The *SatisfiedCost* function returns a lower bound on the cost required to satisfy $\mathcal{G}$ by considering C_{min}.

Because time and resources are probabilistic, the cost and the probability of suc-

Algorithm 3 Evaluate minimum final cost heuristic function

```
1.  function EvaluateMinimumFinalCost(s, G)
2.    C_min(.,.) ← +∞
3.    for each x ∈ X
4.      C_min(x, s.U(x)) ← V(x)
5.    while SatisfiedCost(C_min, G)= +∞
6.      for each a ∈ A
7.        s ← build a state from C_min satisfies a
8.        s' ← Apply(s, a)
9.        for each x ∈ vars(effects(a))
10.         C_min(x, s.U(x)) ← min(C_min(x, s.U(x)), s'.V(x))
11.   return SatisfiedCost(C_min, G)
```

cess of a plan are also probabilistic. Definition 1.3.1 revises the notion of admissibility of a heuristic defined in a domain with time uncertainty.

Definition 1.3.1. A heuristic is *probabilistically admissible* if Equation (1.8) is satisfied for any fixed probability threshold α. The cost of a single execution of an optimal plan is designated by $cost(exec(\pi^*, s_0))$. Hence, the probability that the heuristic does not overestimate the remaining cost of an optimal plan is greater than or equal to α.

$$P\left(\text{EvaluateMinimumFinalCost}(s, \mathcal{G}) \leq cost(exec(\pi^*, s_0))\right) \geq \alpha \tag{1.8}$$

To speed up the evaluation of the heuristic function in Algorithm 3, all calculations are done using scalar values instead of random variables. Hence the domain of the function C_{min} is $\mathbb{R}$ instead of the set of time random variables T. Thus, at Line 4, the time random variable $\mathcal{V}(x)$ is translated into a numerical value. To satisfy the probabilistic admissibility, the inverse of the cumulative distribution function of time random variables is used and denoted as Φ^{-1}. Then, Line 4 is replaced by Equation (1.9). Line 10 is also adapted similarly. The Apply function uses $\Phi^{-1}_{d_{a_x}}(min(\alpha, 0.5))$ as scalar duration.

$$C_{min}(x, \mathcal{U}(x)) \leftarrow \min\left(\Phi^{-1}_{T(x)}(\alpha), E[\mathcal{V}(x)]\right) \tag{1.9}$$

Theorem 1.3.1. The heuristic function presented in Algorithm 3 is probabilistically admissible.

Proof. Algorithm 3 is an adaption of a minimum distance to goal for planning problems with concurrency and uncertainty of continuous values. For fully deterministic planning, this kind of heuristic never overestimates the optimal cost [15]. Because it ignores delete (negative) effects, the cost of the generated relaxed plan is a lower bound on the cost of an optimal plan.

Now consider time uncertainty. As previously said, the heuristic function is evaluated using scalar values rather than random variables. We have to make sure that queries to the inverse of the cumulative distribution function do not overestimate the remaining cost of an optimal plan. Basically, equations of time random variables (T) are expressed using two operators: the **sum** operator to compute the end time of an action, and the **max** operator to compute the start time of an action.

The case of the **sum** operator corresponds to a sequence of actions $a_1, \ldots, a_n$. This generates a time random variable t_{end} defined by the equation $t_{end} = d_{a_1} + \cdots + d_{a_n}$. In that case, Equation (1.10) shows that the heuristic function does not overestimate the optimal final cost.

$$\sum_{i=1}^{n} \Phi_{d_i}^{-1}(min(\alpha, 0.5)) \leq \Phi_{t_{end}}^{-1}(\alpha) \tag{1.10}$$

The case of the **max** operator corresponds to when an action b requires the completion of other actions $a_1, \ldots, a_n$. Therefore, the start time of the action b is defined by a random variable $t_{start} = \max(d_{a_1}, \ldots, d_{a_n})$. In that case, Equation (1.11) shows that the heuristic function does not overestimate the optimal final cost.

$$\max_{i=1}^{n} \Phi_{d_{a_i}}^{-1}(\alpha) \leq \Phi_{t_{start}}^{-1}(\alpha) = \Phi_{\max(d_{a_1},\ldots,d_{a_n})}^{-1}(\alpha) \tag{1.11}$$

The heuristic is thus probabilistically admissible. □

1.3.4 State Kernel Pruning Strategy

Definition 1.3.2. The *kernel of a state* $s = (\mathcal{U}, \mathcal{V}, \mathcal{R}, \mathcal{W})$ is defined by $kernel(s) = (\mathcal{U})$. This corresponds to a world state trimmed of the time information.

Definition 1.3.3. A state s *dominates* another state s' when :

- $kernel(s) = kernel(s')$;
- and $s.V(x) \leq s'.V(x) \forall x \in X$;
- and $s.R(x) \leq s'.R(x) \forall x \in X$;
- and $s.W(y) \leq s'.W(y) \forall y \in Y$.

A state s *strictly dominates* another state s' when at least one of the three $\leq$ can be replaced by $<$ for a state variable $x \in X$ or $y \in Y$.

Theorem 1.3.2. In Algorithm 2, without sacrificing optimality, a state s can be pruned when another already visited state s' dominates s.

1.3.5 Completeness and Optimality

This section discusses about the completeness and optimality of ActuPlan^{nc}. Few assumptions are required to provide some guarantees about these properties. The first assumption is that the planning time horizon has to be finite. This assumption is reasonable because it is possible to fix an upper bound on the maximum duration of the optimal plan. In the presence of deadlines, this upper bound can be set to the latest deadline. Another assumption is that actions have a strictly positive expected duration. Combined together, these assumptions guarantee the state space to be finite.

It has been demonstrated that decision-epoch planners can be incomplete for particular planning domains which require concurrency [24]. Problems requiring concurrency includes domains having action with *at end* conditions. When *at end* conditions require synchronizations with other actions, limiting decision epochs to the end of actions is incomplete [24]. Since this this kind of problems is not addressed in this paper, *at end* conditions are proscribed. Allowed planning domains by ActuPlan^{nc} are less general than full PDDL. However, *at end* conditions are rarely involved in typical planning problems. Moreover, when there is uncertainty on the duration of actions, *at end* conditions are not appropriate.

As presented in Section 1.3.2, ActuPlan relies on an approximate sampling algorithm to estimate continuous random variables. Thus optimality cannot be guaranteed. Definitions 1.3.4 and 1.3.5 introduce the notions of optimality and ϵ-optimality about nonconditional plans.

Definition 1.3.4. A nonconditional plan π^* is optimal for a fixed threshold α on the probability of success when $P(\pi^* \models \mathcal{G}) \geq \alpha$ and no other nonconditional plan π such $P(\pi \models \mathcal{G}) \geq \alpha$ and $E[cost(\pi)] < E[cost(\pi^*)]$ exists.

Definition 1.3.5. A nonconditional plan π^* is ϵ-optimal for a fixed threshold α on the probability of success when $P(\pi^* \models \mathcal{G}) \geq \alpha$ and no other nonconditional plan π such $P(\pi \models \mathcal{G}) \geq \alpha$ and $E[cost(\pi)] + \epsilon < E[cost(\pi^*)]$ exists.

Theorem 1.3.3. ActuPlan^{nc} (Algorithm 2) generates nonconditional plans which are ϵ_1-optimal and has a probability of failure smaller than ϵ_2. Both ϵ_1 and ϵ_2 can be arbitrary small by using a sufficient number of samples m and are computed under a given confidence level γ.

Proof. Algorithm 2 performs a best-first-search which is complete and optimal [57]. However, there is two issues in ActuPlan because an approximate algorithm is involved to estimate random variables. Since (1) the goal satisfaction condition $s \models \mathcal{G}$ and (2) the cost $cost(\pi)$ of the returned plan π are random variables, estimation errors have consequences on completeness, correctness and optimality.

The first consequence is about the completeness and correctness. Since an approximate algorithm is used, condition at Line 2 of Algorithm 2 ($P(s \models \mathcal{G}) \geq \alpha$) may fail to branch correctly. The condition $P(s \models \mathcal{G}) \geq \alpha$ may be estimated to be true while the real probability is smaller than α. The reverse is also possible. A probability of failure ϵ_2 can be estimated since $P(s \models \mathcal{G}) \geq \alpha$ is a Boolean random variable. The error ϵ_2 tends to zero when the number of samples m tends to infinity.

The second consequence is about the optimality. The nondeterministic choices at Line 5 of Algorithm 2 is implemented by an open list in the best-first-search algorithm. The open list requires to be sorted in an increasing order of an evaluation function, which is the cost of plans related to states. Since the costs are random variables, estimation errors on them perturbes the sorting order. This means than the real cost of the first element can be higher than another element in the list. Thus, a non-optimal plan could be returned instead the real one. However, an upper bound ϵ_1 can be estimated on the difference of the cost of the first element and the real minimal cost. This upper bound ϵ_1 is related to the maximum estimation error of the

cost evaluation function of plans. Thus, generated plans are ϵ_1-optimal. The error ϵ_1 tends to zero when the number of samples m tends to infinity. □

1.3.6 Finding Equivalent Random Variables

To prevent an infinite loop, forward-search and state-space-based planning algorithms require to test whether a state s has already been visited. Finding equivalent states is also important to reduce the size of the state space. Recall that two states are equivalent if their mapping functions $\mathcal{U}, \mathcal{V}, \mathcal{R}, \mathcal{W}$ are equivalent. Because functions $\mathcal{V}$ and $\mathcal{R}$ involve time random variables, and $\mathcal{W}$ involves numerical random variables, the notion of equivalence for random variables has to be defined.

Definition 1.3.6. Two random variables are equivalent if and only if they always take the same value.

A simple way to find equivalent random variables could be to compare their equations. Two random variables both described by the same equation implies that they have the same value. However, two time random variables may be equivalent even if their equations are not the same.

Consider a situation where two actions a_1 and a_2 cannot be executed concurrently, and thus have to be executed sequentially. Time t_0 is the initial time. Executing a_1 and a_2 successively leads to times t_1 and t_3. The reverse order leads to times t_2 and t_4. Figure 1.9 shows the Bayesian network for this situation. Equations are shown in the second block of each random variable. Since t_2 and t_4 have different equations, they would be considered as two different random variables, although they clearly both represent the same duration. Thus both states would need to be explored by the planner. To reduce the search space, a better method for testing the equivalence of random variables is required.

Canonical Representation

To find equivalent random variables, it is possible to analytically compare their equations in a canonical form. Thus, two random variables are equivalent if and only if their equations in a canonical form are the same. An equation can be transformed

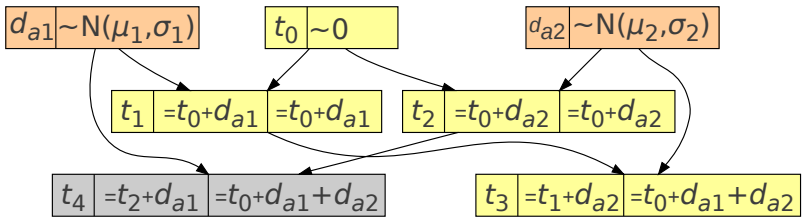


Figure 1.9: Sample Bayesian network with equations in original form (2^{nd} block) and canonical form (3^{rd} block)

in canonical form by only using root random variables. An example is shown in Figure 1.9 where equations displayed in the third block of each random variable are in a canonical form. Because t_3 and t_4 have the same canonical representation, they are equivalent. Thus, t_4 can be eliminated and be replaced by t_3.

Unfortunately a canonical form for equations has a significant cost. The size of an equation in a canonical form for a random variable grows with its depth in the Bayesian network. For instance, the execution of n actions produces n time random variables $t_1, t_2, \ldots, t_n$. The i^{th} random variable has i terms : $t_i = t_0 + d_{a_1} + \cdots + d_{a_i}$. Adopting this strategy costs $\Omega(n \times d)$ in memory and time where n is the number of random variables and d is the average depth.

Comparing Arrays of Samples

To avoid the cost of a canonical representation, a faster method is adopted. The equivalence of random variables is tested by simply comparing their arrays of samples. Remember that only root variables are randomly sampled from their probability distribution function. Samples of other random variables are rather generated by evaluating their equations. This implies that two equivalent random variables will have exactly the same values in their arrays of samples. Figure 1.10 shows the Bayesian network with arrays of samples attached to random variables. The variables d_{a1} and d_{a2} follow the same probability distribution but they are independent, so their generated arrays of samples are different. The variables t_3 and t_4 are equivalent since their arrays of samples are equivalent.

Two random variables having two different arrays of samples is a sufficient condition to assert their non-equivalence. But is the inverse also true? I.e., are two random

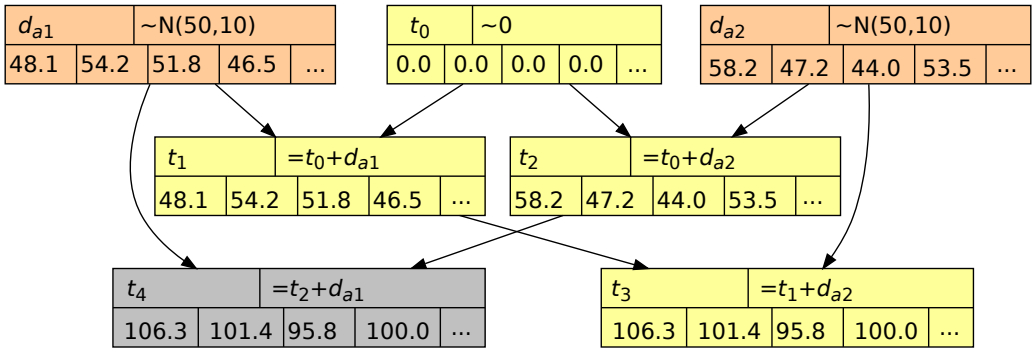


Figure 1.10: Bayesian network with arrays of samples attached to random variables

variables having the same array of samples necessary equivalent? Theoretically, yes because a real value has a zero probability to be sampled. But this is not true in practice because floating-point numbers in computers do not have an infinite precision. To cope with numerical errors, a small ϵ is generally required to compare the equality of two floating-point numbers. Two samples a and b are judged to be equal when $|a - b| \leq \epsilon$.

This means that, in practice, two non-equivalent random variables could potentially generate approximately the same (close to ϵ) array of samples. The probability that two non-equivalent random variables X and Y generate approximately the same arrays of m samples M_X and M_Y is infinitely small. Let consider two cases involving uniform and normal distributions.

Let random variables $X \sim \mathcal{U}(x_1, x_2)$ and $Y \sim \mathcal{U}(y_1, y_2)$ be independent. Two arrays of m samples M_X and M_Y are respectively generated from X and Y. The probability that both arrays are approximately equal (close to an ϵ) is given by Equation (1.12). The *overlaps* function returns the length where invervals X and Y overlap.

$$P(M_X \approx M_Y) = ((x_2 - x_1) \times overlaps(X, Y) \times (y_2 - y_1) \times \epsilon)^m \tag{1.12}$$

This probability is maximized when both random variables totally overlap and when they are on a small interval like $X \sim \mathcal{U}(0, 1)$ and $Y \sim \mathcal{U}(0, 1)$. In that case,

this probability is as infinitely low as 10^{-140} with $\epsilon = 10^{-7}$ and only $m = 20$ samples.

Now consider another case with normal distributions. Let $X \sim \mathcal{N}(\mu_1, \sigma_1)$ and $Y_1 \sim \mathcal{N}(\mu_2, \sigma_2)$ be two independent random variables. The probability of generating approximately the same arrays of m samples M_X and M_Y is given by Equation (1.13).

$$P(M_X \approx M_Y) = [P(|X - Y| \leq \epsilon)]^m \tag{1.13}$$

Let $Z = X - Y$. Thus $Z \sim N(\mu_1 - \mu_2, \sqrt{\sigma_1^2 + \sigma_2^2})$. Equation (1.14) is obtained from Equation (1.13). Φ_Z is the cumulative probability function of the distribution of Z.

$$\begin{aligned} P(M_X \approx M_Y) &= [P(Z \leq \epsilon)]^m \\ &= [\Phi_Z(\epsilon) - \Phi_Z(-\epsilon)]^m \end{aligned} \tag{1.14}$$

The probability of approximate equivalence of samples is maximized when both random variables have the same mean and have low standard deviations. For instance, let X and Y be two random variables which follow $\mathcal{N}(0, 1)$. In that case, this probability is as infinitely low as 1.1×10^{-142} with $\epsilon = 10^{-7}$ and only $m = 20$ samples. These examples confirm that comparing arrays of samples of random variables is a reasonable method to test the equivalence of two random variables.

1.4 ActuPlan : Conditional Plannner

The conditional planner is built on top of the nonconditional planner presented in Section 1.3. The basic idea behind the conditional planner is that the non conditional planner can be modified to generate several nonconditional plans with different tradeoffs on their probability of success and their cost. A better conditional plan is then obtained by merging multiple nonconditional plans and time conditions to control its execution.

1.4.1 Intuitive Example

Before detailing the conditional planner, an intuitive example is presented. Consider the situation presented in Figure 1.11. A truck initially parked in location l_2

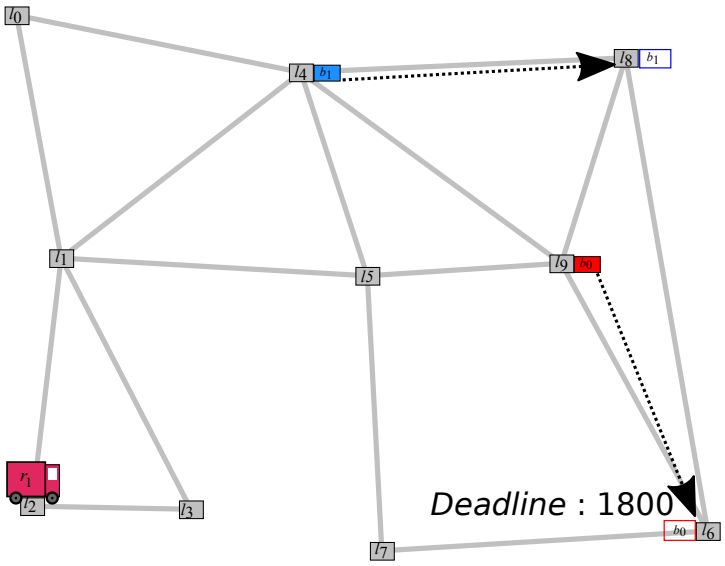


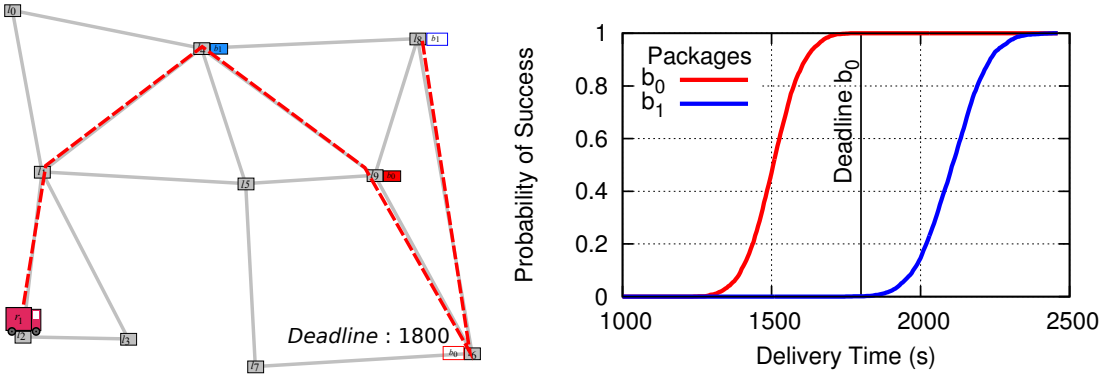
Figure 1.11: Example with two goals

has to deliver two packages b_0 and b_1 from locations l_9 and l_8 to locations l_6 and l_8. The package b_0 destined to location l_6 has time $t = 1800$ as a delivery deadline. The cost is defined by the total duration of the plan (makespan).

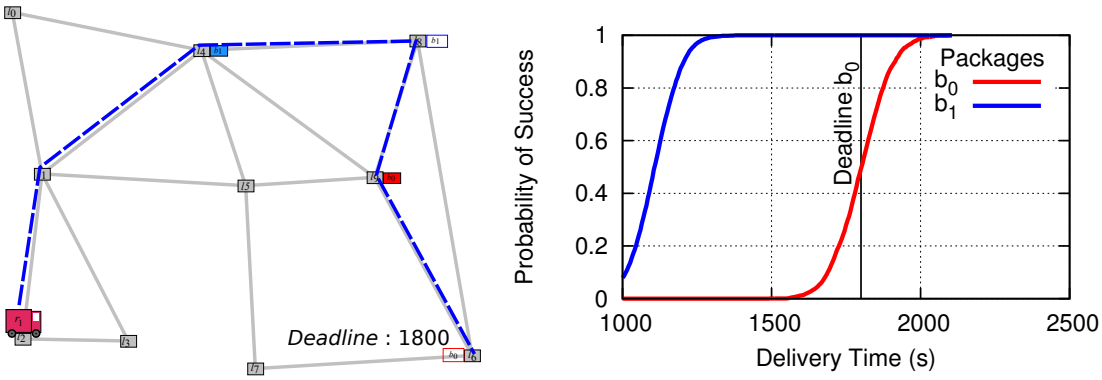
Figure 1.12 shows two possible nonconditional plans. The plan π_a delivers package b_0 first while plan π_b delivers package b_1 first. Since package b_0 has the earliest deadline, plan π_a has an higher probability of success than plan π_b. However, plan π_a has a longer makespan than plan π_b because its does not take the shortest path. If the threshold on the probability of success if fixed to $\alpha = 0.8$, the nonconditional plan π_a must be returned by ActuPlan$^{\text{nc}}$.

A better solution is to delay the decision about which package to deliver first. The common prefix of plans π_a and π_b ($Goto(r_1, l_2, l_4)$, $Load(r_1, l_4, b_1)$) is first executed. Let the resulting state of this prefix be s_2. As illustrated in Figure 1.13, with the respect of nonconditional plans π_a and π_b, there is two possible actions : $Goto(r_1, l_4, l_8)$ or $Goto(r_1, l_4, l_9)$.

Since the duration of the common prefix is uncertain, the decision of which action to take depends on the current time. If the execution of the prefix is fast, action $Goto(r_1, l_4, l_9)$ (delivering package b_1 first) is the best decision. If not, it is preferable to choose action $Goto(r_1, l_4, l_8)$ (delivering package b_0 first). This decision can be guided



(a) Plan π_a



(b) Plan π_b

Figure 1.12: Possible nonconditional plans

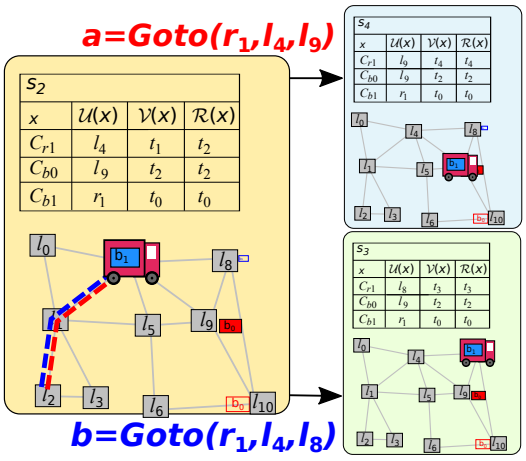


Figure 1.13: Possible actions in state s_2

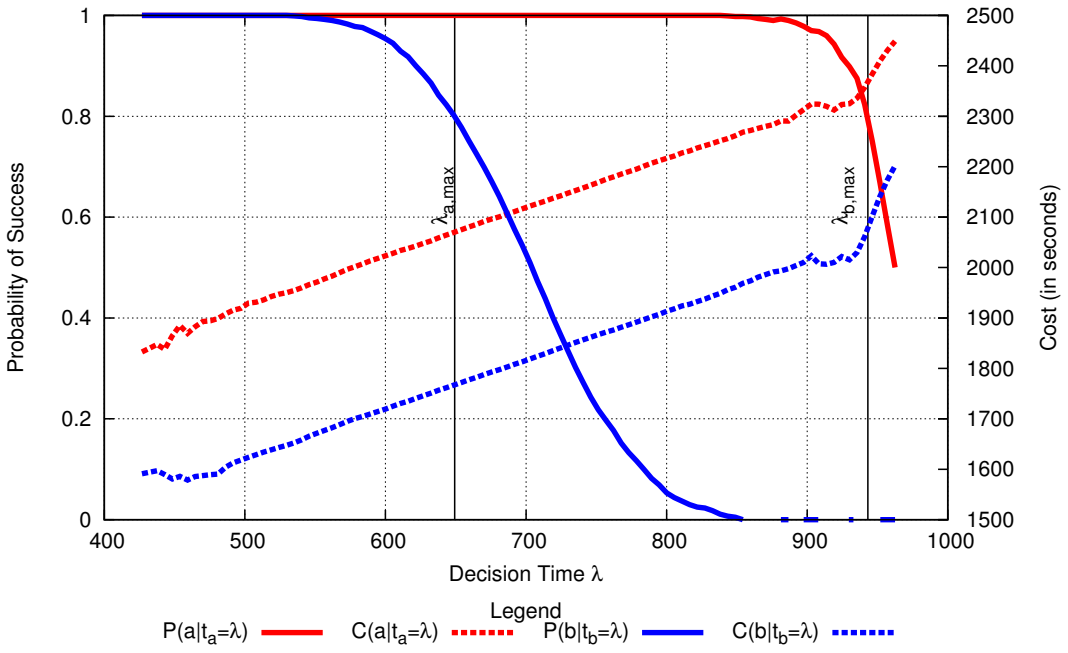


Figure 1.14: Latest times for starting actions

by the latest time at witch choosing action $Goto(r_1, l_4, l_8)$ still satisfies the threshold on the probability of success α. This time can be computed by evaluating the conditional probability of success knowing the decision time at which $Goto(r_1, l_4, l_8)$ is started. Figure 1.14 presents a chart which illustrates how this critical time is computed. Solid lines represent the conditional probability of success of choosing plans π_a and π_b while dashed lines represent the conditional cost. If the threshold on the probability of success is fixed at $\alpha = 0.8$, then the latest time at which the plan π_b can be selected is around time 652.

1.4.2 Revised Nonconditional Planner

The nonconditional planning algorithm presented in Section 1.3 needs to be revised in order to generate nonconditional plans to be merged later into a better conditional plan. Algorithm 4 presents the adapted nonconditional planner in an iterative form. Basically, the algorithm performs a forward best-first-search to generate a set of nonconditional plans, each having a probability of success of at least β. The

nonconditional plans are not explicitly generated. Instead, a search graph is expanded and the algorithm returns the set of final states F reached by the implicitly-generated nonconditional plans. The nonconditional plans will later be merged to satisfy the minimal probability of success α. Each time a state s which satisfies the goal $\mathcal{G}$ is found (Line 9), it is added to the set of final states F. The algorithm terminates when the *open* set is empty, i.e., when all reachable states before the allowed time horizon are visited. If the heuristic is admissible, the algorithm can be also stopped when a state s is visited with a maximum valid time greater than t_α (Line 8). The random variable t_α represents the minimal cost of a nonconditional plan with $P(\pi \models \mathcal{G}) \geq \alpha$ (Line 12). The *parents* attributes contains all parent states from which a state is accessible (see Line 16).

Algorithm 4 Algorithm for generating a set of nonconditional plans

```
1.  GenerateNonCondPlans(s_0, G, A, α, β)
2.    open ← {s_0}
3.    close ← ∅
4.    t_α ←~ N(+∞, 0)
5.    while open ≠ ∅
6.      s ← open.RemoveFirst()
7.      add s to close
8.      if max_{x∈X}(s.V(x)) > t_α exit while loop
9.      if P(s ⊨ G) ≥ β
10.       add s to F
11.       if P(s ⊨ G) ≥ α
12.         t_α ← min(t_α, max_{x∈X}(s.V(x)))
13.     else
14.       for each a ∈ A such a is applicable in s
15.         s' ← Apply(s, a)
16.         add s' to s.parents
17.         s'.f ← EvaluateMinimumFinalCost(s', G, β)
18.         if s' ∉ close and P̄(s' ⊨* G) ≥ β
19.           add s' to open
20.   return F
```

Theorem 1.4.1. If $\beta = 0^+$ then Algorithm 4 returns a set of final states F such that each $f \in F$ is reached by at least one successful execution of an optimal conditional plan.

Proof. Let $\omega = \langle s_0, a_1, s_1, ..., a_n, s_n \rangle$ be a successful execution trace of an optimal

conditional plan. Let $s_n \notin F$, i.e., Algorithm 4 failed to reach s_n. This means that there exists at least one first state s_i such $i > 0$ in the execution trace which has not been visited by Algorithm 4. Since $\beta = 0^+$, the state s_i cannot be pruned at Line 9. This implies that action a_i has not been applied in state s_{i-1}. Since Line 13 applies all actions of the domain, this is not possible. The other possibility is that state s_{i-1} has not been visited. This is a contradiction since s_i is the first state in trace ω which has not been visited. □

By Theorem 1.4.1, an optimal conditional plan can be theoretically built from the expanded graph of Algorithm 4.

Theorem 1.4.2. Let π^* be an arbitrary optimal conditional plan and F the set of final states returned by Algorithm 4. Then, the probability that π^* reaches a state $f \in F$ is at least $1 - \beta$.

Proof. This proof is similar to that of Theorem 1.4.1. Let $\omega = \langle s_0, a_1, s_1, ..., a_n, s_n \rangle$ be a successful execution trace of an optimal conditional plan π^*. Let $s_n \notin F$, i.e., Algorithm 4 failed to reach s_n. This means that there exists at least one first state s_i such $i > 0$ in the execution trace which has not been visited by Algorithm 4. As explained in the previous proof, the only one possibility is that state s_i has been pruned by Line 9. The state can be pruned only if it represents a probability of success lower than β. Missing states in F are those which are reachable from a sequence of actions which has a probability of success lower than β. Thus, Algorithm 4 returns a set of final states F which are reached by a proportion of at least $1 - \beta$ of possible successful executions of an optimal conditional plan π^*. □

Theorem 1.4.2 is a generalization of Theorem 1.4.1 that guarantees the sub-optimality of a conditional plan after a call to GenerateNonCondPlans with $\beta > 0$.

Once the set of final states F is found, Algorithm 5 selects the set of candidate actions $s.\breve{A} \subseteq A$ for each state s reachable by Algorithm 4. A candidate action $a \in s.\breve{A}$ is any action that can be executed at this point in time. Conditional branches are generated by the conditional planner to guide the decision concerning which candidate action to execute in each state.

Algorithm 5 Algorithm for selecting candidate actions

```
1.  SelectCandidateActions(F, A)
2.    open ← F
3.    S ← ∅
4.    while open ≠ ∅
5.      s ← open.RemoveAnElement()
6.      add s to S
7.      add all s' ∈ s.parents to open
8.    for each s ∈ S
9.      for each a ∈ A such a is applicable in s
10.       s' ← Apply(s, a)
11.         if s' ∈ S
12.           add a to s.Ă
```

Algorithm 6 Time conditioning planning algorithm

```
1.  ActuPlan(s_0, G, A, α, β)
2.    F ←GenerateNonCondPlans(s_0,G, A, α, β)
3.    SelectCandidateActions(F, A)
4.    S ← F
5.    while S ≠ ∅
6.      s ← pick a state s ∈ S | s' is visited ∀a ∈ s.Ă, s' = Apply(s, a)
7.      if s ∈ F
8.        s.Success ← s ⊨ G
9.        s.FinalCost ← Cost(s)
10.     else
11.       for each a ∈ s.Ă
12.         s_a ← Apply(s, a)
13.         λ_{a,max} ← find upper bound λ such that E[s_a.Success | t_a = λ] ≥ α
14.       for each a ∈ s.Ă
15.         s_a ← Apply(s, a)
16.         λ_{a,min} ← find lower bound λ such that
17.               λ ≤ λ_{a,max} i.e. E[s_a.Success | t_a = λ] ≥ α
18.               and E[s_a.FinalCost|t_a = λ] ≤ min_{b∈s.Ă} E[Apply(s, b).FinalCost|λ < t_b ≤ λ_{b,max}]
19.         add (a, λ_{a,min}, λ_{a,max}) to π(s)
20.       s.Success ←Apply(s, Decision(s, π(s))).Success
21.       s.FinalCost ←Apply(s, Decision(s, π(s))).FinalCost
22.     add all s' ∈ s.parents to S
23.     mark s as visited
24.   return π
```

1.4.3 Time Conditional Planner

The time conditioning planning algorithm assumes that the only available observations at execution are the current time and the set of applicable actions. An action becomes applicable as soon as the assignments of the state variables involved in the *at start* conditions become valid. The valid time of these assignments might depend on the completion of other actions in progress.

The time conditioning planning algorithm is presented in Algorithm 6. Basically, the algorithm iterates in the graph which has been expanded by the nonconditional planner (Algorithm 4) in reverse order, i.e. from the final states to the initial state. For each visited state s, it computes the best conditions on time as to minimize the expected final cost and to assert a probability of success of at least α, where $\beta \leq \alpha \leq 1$.

The algorithm proceeds as follows. At Line 2, the nonconditional planner is called and returns the set F of final states which satisfy the goal. At Line 3, the function SelectCandidateActions is called to select the set of candidate actions $s.\check{A}$ for each state s. The set S contains states to be visited by the *while* loop of Line 5. It is initialized with the contents of F at Line 4. Line 6 picks a state $s \in S$ such that all successor states s' resulting of an application of a candidate action of s has already been visited by the algorithm. Once the algorithm is in a state s, branching conditions are computed to dictate how to choose the right candidate action at execution time.

Let $s.\check{A} = \{a_1, \ldots, a_n\}$ be the set of candidate actions for state s. The best decision is to execute the action which minimizes the expected final cost with a probability of success greater than or equal to α. However, this choice cannot be instantaneous because the current state does not represent a snapshot of the environment, but rather an indication of how the environment progresses forward in time. The time random variables ($s.\mathcal{V}(x)$) attached to state variables ($x \in X$) model the probabilistic time at which their assignments ($s.\mathcal{U}(x)$) become valid. Some candidate actions in $s.\check{A}$ are enabled while others are not, depending on the completion of previous actions. Moreover, the expected final cost and the probability of success of each choice depends on the time at which the actions are started. Basically, the decision on whether to start an action is based on the following question. If a candidate action a_i is enabled at current time λ, should we start it or should we wait in case another better action

becomes enabled later?

To answer this question, two critical times $(\lambda_{a,min}, \lambda_{a,max})$ are computed for each candidate action $a \in s.\check{A}$. The time $\lambda_{a,min}$ is the earliest time at which an action a must be started if it is enabled. Before time $\lambda_{a,min}$, if the action is enabled, it is preferable to wait in case another better candidate action b becomes enabled. The time $\lambda_{a,max}$ indicates the latest time that an action can be started just before the resulting probability of success drops under α.

A conditional plan π is a mapping function $\pi : S \rightarrow \bigotimes_{a \in s.\check{A}}(a, \mathbb{R}^+, \mathbb{R}^+)$. For a state s, a decision condition $\pi(s)$ expressed as a set of tuples $\pi(s) = \{(a, \lambda_{a,min}, \lambda_{a,max}) \mid a \in s.\check{A}\}$.

These critical times are computed via an analysis of the distribution of random variables in the Bayesian network. Two new random variables are created for each state. The Boolean random variable $s.Success$ represents whether the decisions taken at state s resulted in a successful execution. For final states $s \in F$, it is defined as the satisfaction of deadlines in $\mathcal{G}$ (Line 8). The continuous random variable $s.FinalCost$ represents the total final cost of an execution resulting from the decisions taken at state s. For final states $s \in F$, it is defined as the cost of being in those states (Line 9). For instance, when the cost is the makespan, $s.FinalCost = \max_{x \in X} s.\mathcal{V}(x)\ \forall s\ \in F$.

Branching conditions are computed in the *else* block (Line 10). For each candidate action a, Lines 15-17 compute the earliest time λ at which it is better to execute a than to wait for another action b. An action a is better at time λ when its expected cost is less than the expected cost of starting another candidate action b at time t_b which is constrained by $\lambda < t_b \leq \lambda_{b,max}$. The critical times are used to build the conditional plan (Line 18).

Once the decision policy is made for a state s, it is possible to create the random variables $s.Success$ and $s.FinalCost$. The probability distribution of these random variables is obtained by simulating the outcome corresponding to this decision. The expected cost of the resulting conditional plan is given by $E[s_0.FinalCost]$.

Theorem 1.4.3. Algorithm 6 generates an ϵ-optimal conditional plan π^* when $\beta = 0^+$, $\alpha = 1$ and the set of observations is limited to the current time and the set of applicable actions.

Proof. Algorithm 4 (GenerateNonCondPlans) generates a set of nonconditional plans where each plan represents a path that could be followed by an ϵ-optimal conditional plan. For each final state s, random variables $s.Success$ and $s.FinalCost$ are computed. Their distributions are estimated using the Bayesian network. Up to the estimation error, this represents the best information available.

Once the final states are evaluated, the ascendent states are recursively evaluated. The conditions on time for a state are established by minimizing the expected final cost of the plan and by asserting a probability of success greater than α. This decision is taken by considering the random variables $s.Success$ and $s.FinalCost$ which have been previously estimated. Thus, these conditions on time are the best possible conditions, since they lead to globally-best execution paths. This process is done repeatedly until the initial state s_0 is reached. Therefore, decisions taken at all states are optimal under these conditions. □

Theorem 1.4.4. Algorithm 6 generates a sub-optimal conditional plan when $\beta > 0$ and $\alpha < 1$. A lower bound on the expected cost of an optimal plan π^* is given by Equation (1.15).

$$\overline{E\left[cost(\pi^*)\right]} = \begin{cases} \beta E\left[cost(\pi_{\beta=0^+})\right] + (1-\beta)E\left[cost(\,\pi)\right], & \text{if } \alpha = 1 \\ E\left[cost(\pi_{\beta=0^+})\right], & \text{if } \alpha < 1 \end{cases} \tag{1.15}$$

Proof. Since $\beta > 0$, the GenerateNonCondPlans algorithm may miss some possible execution paths with small probabilities of success, but also with a lower cost. Since these missed possible execution paths are less frequent than β, it is possible to assume that they will be executed at most by a proportion of β. The best possible cost of these execution paths is the cost of the best nonconditional plan $\pi_{\beta=0^+}$ with a non-zero probability of success.

When $\alpha < 1$, the time conditioning might be not optimal. Critical times $\lambda_{a,min}$ and $\lambda_{a,max}$ for an action a are computed by finding the moments when the probability of success of choosing a drops exactly to α. In other words, it guarantees a probability of success of at least α every time an action is started. However, the probability of branching is not considered. For instance, let's say the algorithm has to decide

between two actions a and b in a given state s. The algorithm generates a branching condition "if $(time \leq \lambda)$, choose a". Thus, the probability of success of the resulting conditional plan π is :

$$P(\pi) = P(time \leq \lambda)P(a \mid time \leq \lambda) + P(time > \lambda)P(b \mid time > \lambda) \tag{1.16}$$

If $P(b \mid time > \lambda) > \alpha$, $P(time \leq \lambda) > 0$ and $\alpha < 1$ then $P(\pi) > \alpha$. This means that a lower value λ which satisfies $P(\pi) \geq \alpha$ might exist. In the worst case, when $P(a \mid time \leq \lambda) < \alpha$ for all λ and $P(b \mid time > \lambda) = 1$, the action a will never be chosen. Thus, the best estimate of a lower bound on the cost of an optimal plan is given by simply considering $\pi_{\beta=0^+}$. □

Required Approximations

Theorem 1.4.3 and Theorem 1.4.4 are true when it is possible to assume that the distribution of random variables $s.Success$ and $s.FinalCost$ are perfectly defined for every state s. Since these distributions depend on the time random variables $t \in T$, which must be approximated by a direct sampling inference algorithm, they cannot be perfectly asserted. Thus in practice, the conditional planner in Algorithm 6 has the same limitations as the nonconditional planner. Fortunately, in a similar manner to the nonconditional planner, the error bound can be estimated and minimized by increasing the number of samples.

The estimation of conditional expected values like $E[s_a.FinalCost \mid t_a = \lambda]$ represents another difficulty. Indeed, when continuous values are used, conditions such as $t_a = \lambda$ have to be replaced by $t_a \approx \lambda$. Basically, direct sampling algorithms for Bayesian network inference select the subset of samples which match the evidences. The value of the queried variable is then estimated by considering only this subset of samples. Since the estimation error is in inverse proportion to the square root of the number of samples (see Section 1.3.2), a significantly larger number of samples might be required to get enough samples in the matching subset.

Considering Waiting Time in Decisions

At execution time, some decisions may be postponed and the agent may be required to wait. Even though waiting may be advantageous in a given situation, it may still have an impact on the probability of success and on the expected total final cost. Lines 20–21 of Algorithm 6 hide some details about the waiting process requirements. For instance, let the current state be s. Let candidate action $a \in s.\check{A}$ be enabled at time λ. If $\lambda < \lambda_{a,min}$ the agent is required to wait until time $\lambda_{a,min}$ because another action b might become enabled. In that case, the resulting state $Apply(s, a)$ is different from $Apply(s, Decision(\pi(s)))$ because the first one assumes that action a is started as soon as possible, while the second one may postpone the action. To address this issue, it would be possible to modify the Apply function of Algorithm 1 by changing its Line 5 to $t_{start} \leftarrow \max(t_{conds}, t_{release}, \lambda_{a,min})$.

However, this modification causes another problem. The resulting state of the new Apply function could be a new state which has not been generated by the nonconditional planner and has not been visited by Algorithm 6. Thus, its attached random variables $Success$ and $FinalCost$ may not be available.

Algorithm 7 Simulate decision algorithm

```
1.  SimulateDecision(s, π(s))
2.    for each i ∈ {1, 2, 3, ..., m}
3.      for each a ∈ s.Ǎ
4.        t_conds,a,i ← max_{x∈vars(conds(a))} m_{s.V(x),i}
5.        t_release,a,i ← max_{x∈vars(effects(a))} m_{s.R(x),i}
6.        t_ready,a,i ← max(t_conds, t_release)
7.        t_start,a,i ← max(t_conds, t_release, λ_{a,min})
8.      Ã ← {a ∈ s.Ǎ | t_start,a,i ≤ λ_{a,max}}
9.      bestAction ← arg min_{a∈Ã} t_start,a,i
10.     s' ←Apply(s, bestAction)
11.     if t_start,a,i ≤ λ_{bestAction,min}
12.       m_{s.Success,i} ← m_{s'.Success,i}
13.       m_{s.FinalCost,i} ← m_{s'.FinalCost,i}
14.     else
15.       M ← {j ∈ {1, 2, 3, ..., m} | t_start,a,i ≈ t_ready,a,j}
16.       j ← randomly select j ∈ M
17.       m_{s.Success,i} ← m_{s'.Success,j}
18.       m_{s.FinalCost,i} ← m_{s'.FinalCost,j}
```

This problem is solved by simulating the impacts of the decision on the $Success$

and $FinalCost$ random variables. Lines 20–21 of Algorithm 6 are replaced by a call to the *SimulateDecision* function of Algorithm 7, which works as follows. Line 2 iterates on all m independent samples generated by the Bayesian network. For each candidate action a, Lines 3–7 compute the ready time and the start time. The ready time is the earliest time at which an action is enabled while the start time is the earliest time an action can be started at, according to the current plan node $\pi(s)$. Line 8 selects the subset $\tilde{A}$ of candidate actions which assert a probability of success of at least α. Line 9 then simulates the decision about which action to start. The resulting state s' is computed at Line 10. Note that this state does not consider the potential waiting time until $t_{start,a,i}$. However, since this state has previously been visited by Algorithm 6, its attached random variables $s'.Success$ and $s'.FinalCost$ are available. If the decision does not involve waiting (Line 11), we can conclude that the corresponding samples of $s'.Success$ and $s'.FinalCost$ are correct. Thus, samples are simply copied (Lines 12–13). Otherwise, the samples cannot be copied because samples attached to s' are based on the assumption that the action has been started at its ready time. In that case, all samples which are close enough to the real start time are selected at Line 15. Then, a sample is randomly chosen (Line 16) from this set and its success and cost values are used (Lines 17–18).

Example of Time Conditioning

Figure 1.15 shows an example of time conditioning in the state s_2 of Figure 1.13 where actions $a = Goto(r_1, l_4, l_9)$ and $b = Goto(r_1, l_4, l_8)$ are possible. The curves $P(a \mid t_a = \lambda)$ and $P(b \mid t_b = \lambda)$ represent respectively the probability of success of choosing action a and b, i.e., $E[s_a.Success \mid t_a = \lambda]$ and $E[s_b.Success \mid t_b = \lambda]$. Times $\lambda_{a,max}$ and $\lambda_{b,max}$ are the latest times at which actions a and b can be started. These limits are computed by finding for which values $P(a \mid t_a = \lambda)$ and $P(b \mid t_b = \lambda)$ drop under the given threshold on the probability of success $\alpha = 0.8$.

The curves $C(a \mid t_a = \lambda)$ and $C(b \mid t_b = \lambda)$ represent respectively the expected total final cost of choosing and starting the action a at time t_a and the action b at time t_b. The time $\lambda_{a,min}$ is the earliest time at which it becomes advantageous to start action a instead of waiting for action b to become enabled, i.e., $C(a \mid t_a = \lambda) \leq C(b \mid \lambda < t_b \leq \lambda_{b,max})$. The time $\lambda_{b,min}$ is computed similarly. The action b

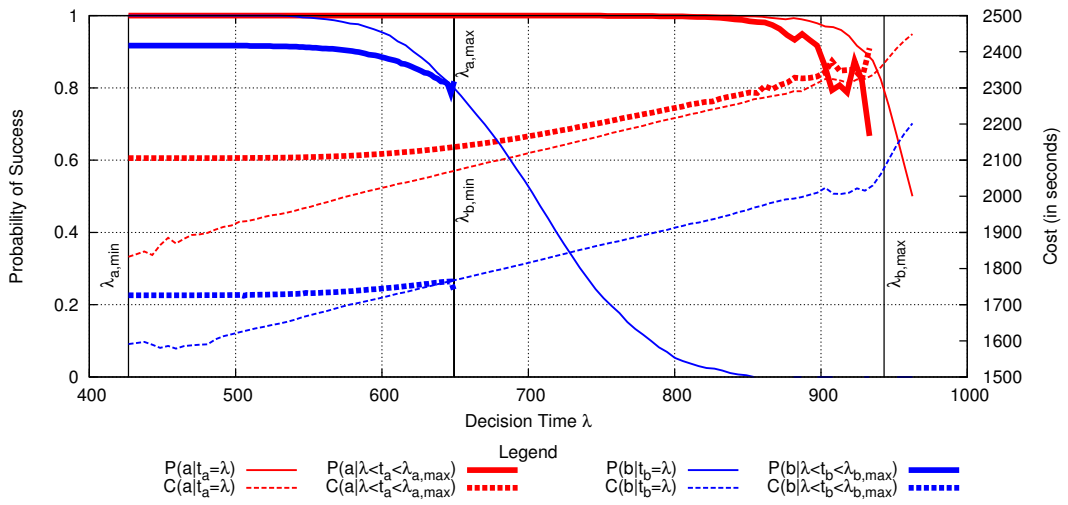


Figure 1.15: Example of time conditioning

becomes the best choice as soon as the probability of success of action a drops below α. Consequently, the time conditioning for state s_2 is $\pi(s) = \langle (a, 425, 652)(b, 652, 971) \rangle$. At execution time, the first action to be enabled in its validity time interval will be started. Figure 1.16 presents the complete generated conditional plan.

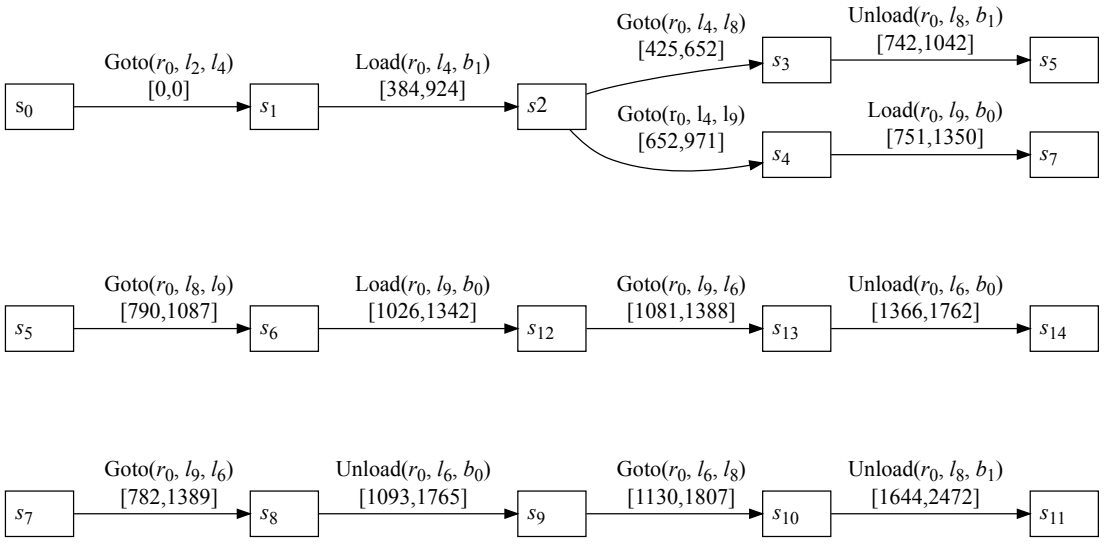


Figure 1.16: Example of conditional plan

1.5 Experimental Results

We experimented ActuPlan planners on the Transport and the Rovers planning domains inspired by the International Planning Competition (IPC) [27]. As presented in Table 1.1, uncertainty has been introduced in the duration of actions. For instance, *Goto* actions have a duration modelled by normal distributions. Time constraints have also been added to goals. For instance, a package may have both a ready time at the origin location and a deadline to be delivered at destination.

ActuPlan planners have been compared to a concurrent MDP-based planner. A comparison with other approaches such GTD [64] was not possible [5]. The experiments were conducted on an Intel Core 2 Quad 2.4 GHz computer with 3 GB of RAM running a 32-bit Linux Kernel version 2.6. The planners were implemented in Java and the experiments were run under the Open JDK virtual machine. A package containing all the source code of ActuPlan planners and the planning problems is available online at http://planiart.usherbrooke.ca/∼eric/quanplan/.

1.5.1 Concurrent MDP-based Planner

The concurrent MDP-based planner is inspired by the work of Mausam and Weld [47]. Their planners are based on the Labeled Real-Time Dynamic Programming (LRTDP) [16] algorithm to solve a decision problem into an interwoven state-space. The involved interwoven state-space is slightly different from the classic approach of forward-chaining for concurrency planning as in TLPlan [2]. An interwoven state $s = (X, Y)$ is defined by a set of assigned state variables X and a set of pairs $Y = \{(a_1, \delta_1), \ldots, (a_n, \delta_n)\}$ which represent the current actions in execution. A pair $(a_i, \delta_i) \in Y$ means that the action a_i is running and has been started δ_i units of time ago. The set Y is similar to the queue of delayed effects in TLPlan.

This original formulation of an interwoven state does not support time constraints since it does not contain any reference to the initial time. To enable time constrains in the interwoven state-space, a timestamp variable t is added to the state representation. Thus, an interwoven state is defined by $s = (X, Y, t)$. The resulting state-space is

5. Public release of GTD is not yet available, based on information obtained from a personal communication with the authors.

thus significantly larger than the original formulation. The transition function is also modified in order to increment the current time t synchronously with δ_i.

The concurrent MDP-based planner requires that timestamps be aligned. Hence, in the presence of continuous action durations, values are rounded to the nearest alignment timestamp. The granularity of this approximation offers a trade-off between the accuracy and the state space size, which in turn affects the planning performances. In the reported experimentations, timestamps are aligned to 30 units of time.

1.5.2 Evaluation of ActuPlan^{nc}

Table 1.2 reports the empirical results. The first and second columns show the size of problems, expressed in terms of the number of trucks and packages for the Transport domain, and in terms of number rovers and data to collect for the Rovers domain. The columns under ActuPlan^{nc} detail the number of states generated, the number of random variables added to the Bayesian Network, the CPU time (in seconds), the estimated expected probability of success and cost (makespan) of plans, and the absolute estimation error under a 95 % confidence level. To estimate the belief of a random variable of the Bayesian Network, 4096 samples are generated. We keep arrays of samples in memory for at most 5000 random variables. The columns under Concurrent MDP-based planner indicate the number of states, the CPU time, the probability of success and the expected cost (makespan). A few tests failed because they reached the allowed CPU time limit (300 seconds) or the maximum number of states (1 and 2 millions).

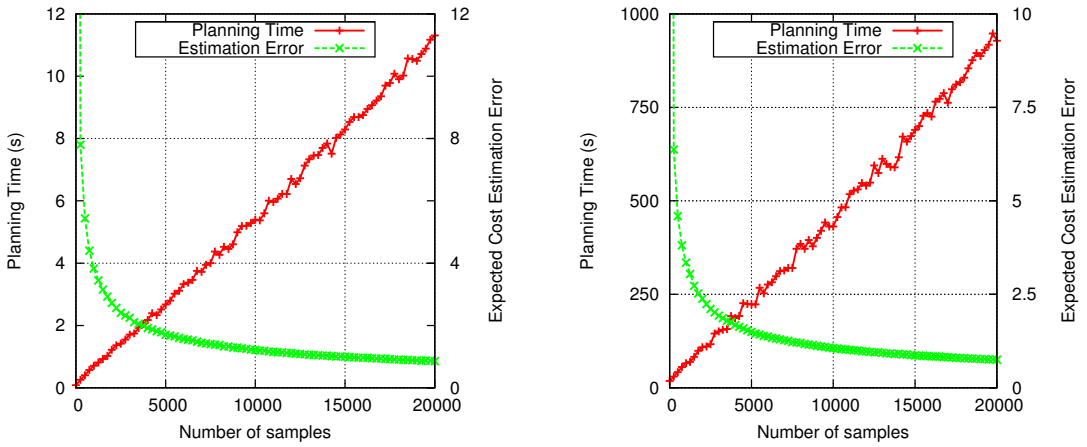
These experiments validate our hypothesis that the overhead of managing random variables is largely compensated by the state space reduction induced. Our approach efficiently avoids the state space explosion caused by the discrete model of time. All solved problems by ActuPlan^{nc} have a probability of success close to 1.0. For the concurrent MDP planner, the probability of success is estimated after the generation of a policy. Since RTDP is an anytime algorithm, the returned policy may have a probability lower than 1.0 when the maximum planning time is reached.

Table 1.2: Empirical results for Transport and Rovers domains

Transport		ActuPlan^{nc}						Concurrent MDP-based Planner			
$\|R\|$	$\|\mathcal{G}\|$	$\|S_{exp}\|$	$\|BN\|$	CPU	P	$Cost$	ϵ	$\|S_{exp}\|$	CPU	P	$Cost$
1	1	6	11	0.3	1.0	715.5	1.4	251	0.0	1.0	715.3
1	2	13	22	0.3	1.0	763.9	1.4	28,837	0.4	1.0	771.9
1	3	32	57	0.3	1.0	1112.1	1.7	1,275,836	32.5	1.0	1124.3
1	4	78	131	0.3	1.0	1604.6	2.0	2,000,174	57.2	0.0	-
2	2	47	56	0.3	1.0	719.2	1.3	953,216	36.2	1.0	719.3
2	3	113	107	0.2	1.0	1013.4	1.5	2,000,399	62.3	0.0	-
2	6	9,859	3,982	5.6	1.0	1084.4	1.4	2,000,821	58.5	0.0	-
2	7	181,740	55,752	132.4	1.0	1338.8	1.8	2,000,975	63.3	0.0	-
2	8	397,370	103,879	264.0	1.0	1339.0	1.8	2,001,088	65.4	0.0	-
3	1	57	46	0.3	1.0	715.9	1.4	554,841	114.2	1.0	697.5
3	4	5,951	2,173	2.5	1.0	980.5	1.8	2,000,633	92.3	0.0	-
3	5	24,194	7,602	9.2	1.0	983.7	1.9	2,000,597	76.3	0.0	-
3	6	90,958	21,938	37.2	1.0	983.0	1.9	2,000,511	82.9	0.0	-
3	7	564,761	80,763	300.0		-		2,001,032	97.6	0.0	-
4	1	267	121	0.2	1.0	717.8	1.3	2,000,038	147.9	0.1	3623.3
4	4	78,775	15,595	32.8	1.0	985.0	1.8	2,000,636	98.5	0.0	-
4	5	474,995	71,367	209.8	1.0	985.8	1.7	2,000,871	110.2	0.0	-
4	6	785,633	96,479	300.0		-		2,000,690	120.4	0.0	-
4	7	803,080	83,077	300.0		-		2,001,063	108.6	0.0	-
4	8	1,000,063	73,181	262.1		-		2,001,237	110.8	0.0	-
Rovers		ActuPlan^{nc}						Concurrent MDP-based Planner			
$\|R\|$	$\|\mathcal{G}\|$	$\|S_{exp}\|$	$\|BN\|$	CPU	P	$Cost$	ϵ	$\|S_{exp}\|$	CPU	P	$Cost$
1	2	50	49	0.073	1.0	1,653	2.1	18,297	3.42	1.0	1,653
1	3	109	80	0.111	1.0	1,966	2.2	142,835	50.3	1.0	1,966
1	4	330	154	0.300	1.0	2,666	2.4	51,765	239	0.1	2,666
1	5	406	156	0.353	1.0	2,892	2.7	-	-	-	-
2	4	6,018	431	1.55	1.0	1,963	2.2	341,937	186	1.0	2,013
2	6	12,445	423	9.72	1.0	2,515	2.4	-	-	-	-
2	8	85,830	1,312	224	1.0	4,571	4.7	-	-	-	-

Impact of the Number of Samples

The necessary use of an inference algorithm to evaluate random variables in the Bayesian network imposes a computational overhead. Direct sampling algorithms have a $\mathcal{O}(nm)$ runtime where n is the number of random variables and m is the number of samples. A higher number of generated samples produces a lower estimation error on the belief of random variables. Figure 1.17 presents the planning time and the estimation error of the plans' cost with respect to the number of samples, for two Transport problems of different size. The planning time grows linearly with the number of samples while the estimation error is in inverse proportion to the square root of the number of samples. For large problems, 4000 to 5000 samples represent a good trade-off between planning speed and the estimation error.



(a) Small Transport Problem (b) Large Transport Problem

Figure 1.17: Impact of number of samples

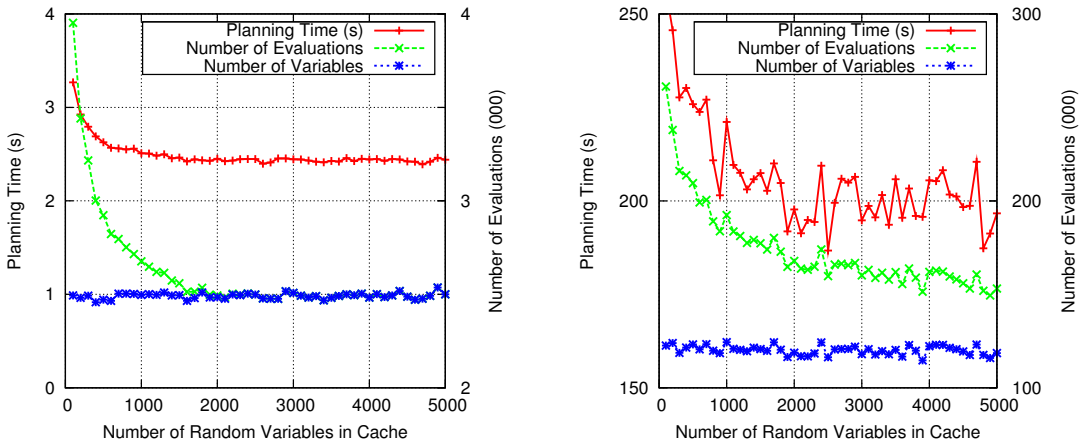
Impact of the Size of Samples Cache

Figure 1.18 presents the impact on planning time induced by a variation on the number of random variables for which the arrays of samples are cached. The advantages of this strategy tend to wear off as the size of the cache increases. For large problems, a few thousand cached random variables offer a good trade-off between memory usage and planning time.

1.5.3 Evaluation of ActuPlan

Table 1.3 presents results for both the nonconditional and the conditional planners. The nonconditional planner was run with $\alpha = 0.9$. The conditional planner was run with $\alpha = 0.9$ and $\beta = 0.4$. These planners are compared to the Concurrent MDP planner. The first and second columns show the size of the problems, expressed in terms of the number of trucks and packages for the Transport domain. Columns under each planner report the CPU time, the probability of success and the cost (makespan). Results show that the conditional planner reduced the expected makespan for most of the problems, when it is possible. For few problems, there does not exist conditional plan which is strictly better than the nonconditional one.

Generated conditional plans are generally better plans but require much more



(a) Small Transport Problem (b) Large Transport Problem

Figure 1.18: Impact of cache size

Table 1.3: Empirical results for the ActuPlan on the Transport domain

Size		ActuPlan^{nc}			ActuPlan			MDP Planner		
$\|R\|$	$\|\mathcal{G}\|$	CPU	P	$Cost$	CPU	P	$Cost$	CPU	P	$Cost$
1	2	0.3	1.00	2105.5	0.3	0.96	1994.5	44.9	0.99	1992.4
1	3	0.3	1.00	2157.3	0.6	0.93	1937.9	214.5	0.00	-
1	4	0.3	0.98	2428.0	0.5	0.98	2426.2	192.6	0.00	-
2	2	0.2	1.00	1174.0	0.4	1.00	1179.2	302.3	0.00	-
2	3	0.3	1.00	1798.3	1.8	0.93	1615.2	288.3	0.00	-
2	4	0.3	1.00	1800.0	4.2	0.91	1500.5	282.8	0.00	-
3	6	4.1	0.97	1460.1	300	0.97	1464.5	307.9	0.00	-

CPU time than ActuPlan^{nc}. This time is required because an optimal conditional plan may require the combination of many plans, which requires to explore a much larger part of the state space. In practice, the choice between ActuPlan^{nc} and ActuPlan should be guided by the potential benefit of having smaller costs. There is a trade-off between plans quality and required time to generate them. When a small improvement on the cost of plans has a significant consequence, then ActuPlan should be used. If decisions have to be taken rapidly, then ActuPlan^{nc} is more appropriate.

1.6 Related Works

Generating plans with actions concurrency under time uncertainty has been recognized as a challenging problem [17]. Even though this particular kind of problem has gained interest over the last decade, little work has been done in the field. Moreover, contributions addressing the problem are often based on different sets of assumptions and are thus difficult to compare. To provide a better overview and to better position the contribution of each approach, Figure 1.19 presents a classification of planning problems related to concurrency and uncertainty. ActuPlan which is presented in this paper is highlighted with a yellow star. Here, in this classification, a distinction is made between two types of uncertainty: the uncertainty on the outcomes of actions, and the numerical uncertainty.

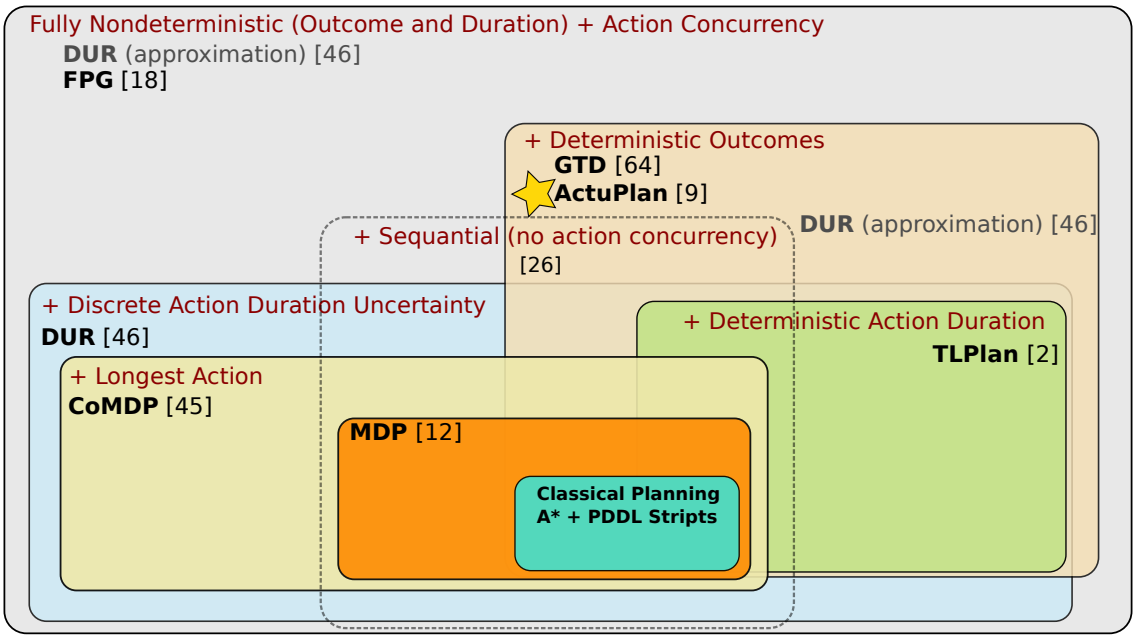


Figure 1.19: Classification of planning problems with actions concurrency and uncertainty

Generally, the more assumptions are made on planning domains, the simpler it is to solve problem instances. For that reason, the classical planning class was the first one to be addressed by the planning community. This very constrained class results from making several assumptions including: (1) the outcomes of actions are deter-

ministic, (2) actions have unit duration (time implicit), and (3) generated plans are sequential (no concurrency). Forward state-based search such as A* and STRIPS [29] are examples of planning algorithms targeting planning domains of that class.

Since this era, a lot of work has been done to progressively relax these assumptions in order to solve more realistic planning problems. The long term goal is to develop a general planner for solving planning problems with any characteristic. Those planning problems are represented by the largest class in Figure 1.19. It contains all planning domains that are fully nondeterministic and allow action concurrency. Full nondeterminism stands for general uncertainty. This class is very general and is rarely addressed directly because problems of this class are very complex. FPG [18] is a planner able to address this kind of problems. It is based on policy-gradient methods borrowed from reinforcement learning [61]. To reduce computation efforts for solving an MDP, policy-gradient methods introduce an approximation function to estimate the states' value during policy generation. Gradients are evaluated using Monte Carlo simulations. To enable concurrency, FPG generates a global factorized policy composed by local policies, each one controlling a specific action.

Most of the existing planning methods cannot address the general class and require more assumptions. In Figure 1.19, each sub-class adds a constraint (planning assumption) on allowed planning domains. On the right, the assumption that actions outcomes are deterministic is added, but actions durations remain uncertain. This class is addressed by the Generate, Test and Debug paradigm (GTD) [64]. The insertion points can be randomly chosen [64] or selected using planning graph analysis [26].

On the bottom, an extra constraint is discrete uncertainty. This requires having actions with a finite number of outcomes and a finite number of durations. For instance, an action could have a duration of either 50, 60 or 70 units of time. CPTP [46] can solve these domains. It is possible to simplify the problem by adding another constraint that fixes the duration of a combined action to the duration of the longest sub-action. The resulting class is the one corresponding to CoMDP [45], the predecessor of CPTP. There exist other MDP-based approaches including Prottle [43], which is based Labeled Real-Time Dynamic Programming (LRTDP) [16], and GSMDP [56].

1.7 Conclusion and Future Work

This paper presented ActuPlan, which is based a new approach for planning under time and resources constraints and uncertainty. The main contribution is a new state representation which is based on a continuous model of time and resources. Continuous random variables are used to model the uncertainty on the time and resources. Time random variables define the start and end times of actions, as well as their duration. The belief of numerical resources is also modelled using random variables. Since a continuous model is more compact than discrete representations, it helps avoid the state space explosion problem.

The random variables are organized into a Bayesian network, a well-established framework in AI to reason about uncertainty. Relying on a Bayesian network offers several advantages. For instance, it offers great flexibility for the possible assumptions about the dependence (or independence) of the random variables related to the duration of actions and to the consumption of resources. This kind of flexibility is sometimes required in real-world applications.

A direct sampling algorithm is used to estimate the probability of success and the expected total cost of plans. We showed that the generated nonconditional plans can be merged to build a conditional plan which produces shorter makespan without sacrificing the probability of success of the merged plans. The test conditions on the branches of the plans are computed through an analysis of the distribution of time random variables.

Two planning algorithms have been presented. The nonconditional algorithm (ActuPlan^{nc}) produces ϵ-optimal nonconditional plans by performing a forward-search in the state space. A minimum final cost heuristic is used to guide the forward-search. The ϵ error can be minimized by increasing the number of samples in the Bayesian network. A time conditional planning algorithm (ActuPlan) generates ϵ-optimal or sub-optimal conditional plans. The conditional plans are obtained by merging several nonconditional plans having different trade-offs between their quality and probability of success. The execution of these plans is guided by conditioning the current time. Depending on the duration of previous actions, the plan chooses the appropriate branches. Empirical results showed the efficiency of our approach on

planning domains having uncertainty on the duration of actions.

The presented approach focuses on numerical uncertainty, i.e., the duration of actions and the consumption of resources, such as energy. We plan to extend our approach to a larger definition of uncertainty including also the uncertainty on the outcomes of actions. The resulting planning algorithm will be a hybrid solution which combines two well established frameworks in AI to deal with uncertainty: an MDP could be used to address this form of uncertainty while numerical uncertainty would be addressed by a Bayesian network.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds québécois de la recherche sur la nature et les technologies (FQRNT).

Chapitre 2

QuanPlan : un planificateur dans un espace d'états quantiques

Résumé

Cet article présente QuanPlan, un planificateur hybride pour la planification d'actions concurrentes sous incertitude générale. Pour résoudre le défi de la concurrence sous incertitude, QuanPlan effectue une recherche dans un espace d'états quantiques. La notion d'état quantique est inspirée de la physique quantique. Un état quantique est une superposition d'états qui permet de modéliser les effets indéterminés des actions. Une nette distinction est faite entre deux formes d'incertitude, soit (1) celle sur le temps (durée des actions) et (2) celle sur les effets des actions. Une approche hybride est présentée. L'incertitude sur la durée des actions est prise en charge par un réseau bayésien construit dynamiquement, tel que présenté au chapitre 1. L'incertitude sur les effets, autres que la durée des actions, est prise en charge par un processus décisionnel markovien (MDP). Le calcul des valeurs des états nécessite des requêtes au réseau bayésien. Le planificateur hybride est validé sur les domaines du Transport et des Rovers.

Commentaires

L'article sera soumis au journal *Artificial Intelligence* (AIJ). Une version plus courte a été soumise à la vingt-cinquième conférence de l'*Association for the Advance of Artificial Intelligence* (AAAI-2011). Le planificateur présenté, QuanPlan, est une généralisation de l'approche présentée dans le premier chapitre. En plus de générer l'incertitude au niveau de la durée des actions, QuanPlan prend également en charge l'incertitude liée aux effets (*outcomes*) des actions. L'article a été rédigé par Éric Beaudry sous la supervision de Froduald Kabanza et François Michaud.

QuanPlan : A Quantum State-Space Planner for Concurrent Actions under Time and Outcome Uncertainty

Éric Beaudry, Froduald Kabanza
Département d'informatique, Université de Sherbrooke,
Sherbrooke, Québec, Canada J1K 2R1
eric.beaudry@usherbrooke.ca, froduald.kabanza@usherbrooke.ca
François Michaud
Département de génie électrique et de génie informatique,
Université de Sherbrooke,
Sherbrooke, Québec, Canada J1K 2R1
francois.michaud@usherbrooke.ca

Abstract

This paper presents QuanPlan, a hybrid planner for planning concurrent actions under uncertainty and time constraints. QuanPlan performs a search in a quantum state space where a quantum state is a superposition of states. Quantum states are used to model the fact that the outcomes of actions are undetermined until their end. A superposition of states is determined (collapsed) when it is observed. Two forms of uncertainty are considered: on the duration of actions and on the outcomes of actions. A continuous model is adopted for the duration of actions. The time of events, corresponding to the start time and the end time of actions, and the duration of actions are represented by continuous random variables. Dependencies are modelled using a dynamically generated Bayesian network. The uncertainty on the outcomes of actions is handled by using a Markov Decision Process. This combination of Bayesian network and Markov Decision Process results in an efficient planner as empirical evaluations on the Transport and the Rovers domains demonstrate.

2.1 Introduction

Planning concurrent actions under uncertainty has been recognized as a challenging problem in AI [17]. This kind of problems is motivated by several real-world applications, such as robotics. For example, the task planning for Mars rovers is an application involving concurrency and uncertainty [17]. A Mars rover has to go to several locations to collect data using specialized sensors. While navigating (which is a task with a probabilistic duration), the rover may perform other tasks like warming up and initializing its sensors, and transmitting the collected data to the base station. Task planning for Mars rovers must account for uncertainty in order to maximize their efficiency, because it is a key aspect of most actions they can perform.

Combining actions concurrency under a general form of uncertainty represents a challenge for planners based on state space exploration. This challenge is particularly present in the state transition function because the actual outcome of an action is only determined at its end. A naïve and approximate approach for implementing actions concurrency under uncertainty would be to implement a state transition function which generates a set of determined states. For instance, the application of a nondeterministic action a from state s would produce the set of successor states $S_a = \{s_1, \ldots, s_n\}$. When concurrency is involved, a state does not necessary have a unique time; it depends on how concurrency is handled. In TLPlan [2] for example, the concurrency is handled by a pending queue of delayed effects. When an action is applied, the current time variable is not increased; the *at end* effects are posted in a queue. In timeline-based planners [9, 52], each state variable is attached to a particular time variable. Thus, being in a resulting state $s' \in S_a$ gives information about the actual outcome of action a. From state s', starting another action b at the same time as action a is a form of "cheating" because the decision to start b is implicitly made under the assumption that the outcome of a is already known. However, since action a is running in state s', its outcome cannot be determined. Consequently, a state-based planner has to deal with a form of undetermined states.

Markov Decision Processes (MDPs) [12] provide a rich and general framework for artificial intelligence (AI) to solve stochastic problems. In its original formulation, an MDP is well-suited for sequential decision-making under uncertainty where exactly

one action (decision) is selected in each state. This framework has been extended to solve problems involving concurrent actions with probabilistic durations and nondeterministic outcomes [47]. This approach adopts an interwoven state space and relies on a single and general model to represent all forms of uncertainty. Indeed, the uncertainty on the duration of actions and on the nondeterministic outcomes are represented as state transitions. Concurrency is handled in a similar fashion to delayed effects in TLPlan [2]. Basically, an interwoven state is the cross product of a classical state and a set of executing actions. Formally, an interwoven state is defined by $s_{\equiv} = (X, \Delta)$, where X is a set of state variables and $\Delta = \{(a_1, \delta_1), \ldots, (a_1, \delta_1)\}$ contains the current actions in execution. A pair $(a, \delta) \in \Delta$ means that action a has been started δ units of time ago. A complex state transition probability function advances the δ time for all running actions at the same time. However, using a discrete representation of time exacerbates the state-space explosion problem.

Simulation-based planning approaches represent an interesting alternative to produce sub-optimal plans. One of them is the Generate, Test and Debug (GTD) paradigm [64], which is based on the integration of a deterministic planner and a plan simulator. In this approach, an initial plan is generated without taking uncertainty into account, and is then simulated using a probabilistic model to identify potential failure points. The plan is incrementally improved by successively adding contingencies to it in order to address uncertainty. The Factored Policy Gradient (FPG) [18] is another planning approach based on policy-gradient methods borrowed from reinforcement learning [61]. However, the scalability of these approaches is limited.

Another approach, which rather relies on a continuous time model, has been proposed to deal with action concurrency under uncertainty on the duration of actions [9] and on numerical effects [10]. The occurrence time of events, which corresponds to the start and end time of actions, and the duration of actions are represented using continuous random variables. The planning algorithm performs a forward-search in a state space where state variables are associated to time random variables. The use of time random variables reduces the size of the state space compared to approaches using a discrete time model. The relationship between these random variables is expressed using a Bayesian network dynamically generated during the search. Queries

are made to the Bayesian network to (1) evaluate the probability that the goal is achieved in a given state of the space explored so far, and to (2) estimate the expected cost (e.g., makespan) of a plan. The planner produces nonconditional plans which minimize the expected cost and assert a probability of success of at least a fixed threshold. However, this approach deals with only one form of uncertainty, i.e., on the time and on numerical resources.

This paper presents QuanPlan, a hybrid planner for planning with actions concurrency under uncertainty. QuanPlan is based on two well established probabilistic frameworks in AI, Bayesian networks and MDPs. These frameworks are used to address respectively two forms of uncertainty: (1) the duration of actions, and (2) the outcomes of actions. This distinction is justified by the fact that in real-world applications, the time is generally continuous while nondeterministic outcomes are generally discrete. The uncertainty on the occurrence of events, i.e., the start and end time of actions, is modelled using continuous random variables [9]. These continuous random variables, which will be named *time random variables* in the rest of the paper, are organized in a dynamically-generated Bayesian network. A direct sampling inference algorithm is used to estimate the distribution of time random variables.

To address the challenge of concurrency under uncertainty, QuanPlan performs a search in a quantum state space. Basically, a quantum state is a superposition of states. Quantum states, i.e., superpositions of states, are required because the actual outcomes of actions are only determined when they end. As in quantum physics, a superposition of states is determined (collapsed) when it is observed. The uncertainty on the nondeterministic outcomes of actions is addressed by an MDP. Nondeterministic actions are modelled by deterministic transitions in a quantum state space. The observation of state variables related to uncertain outcomes cause nondeterministic transitions in the quantum state space.

The rest of the paper is organized as follows. Section 2.2 presents the required basic concepts, and determined and quantum states. Section 2.3 presents the QuanPlan planning approach. Finally, empirical results are reported in Section 2.4 before concluding the paper.

2.2 Basic Concepts

Basic concepts are illustrated using the Mars Rovers domain, which is a benchmark domain [1] from the International Planning Competition of the International Conference on Automated Planning and Scheduling. Simplifications are made to keep the illustrations simple. In that domain, a set of rovers have to gather data from different locations, using a specialized sensor, and transmit the results to a base station using a wireless channel. Figure 2.1 shows a sample map where two robots have to acquire data from locations on a star.

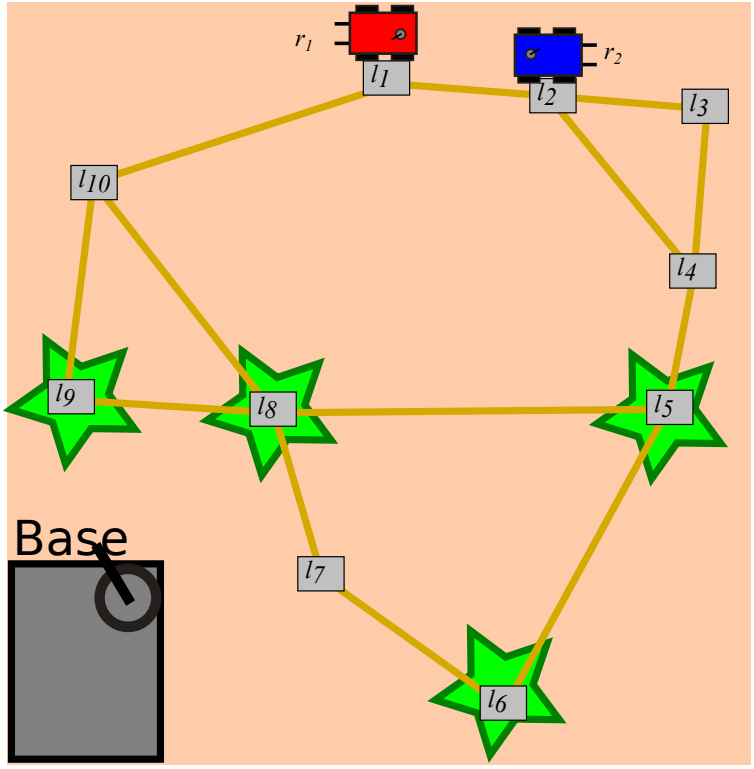


Figure 2.1: Example of a map to explore

1. http://planning.cis.strath.ac.uk/competition/domains.html

2.2.1 State Variables

World states are modelled using a set of discrete *state variables* $X = \{x_1, \ldots, x_n\}$. A state variable $x \in X$ has a finite domain $Dom(x)$. The Rovers planning domain involves a set of n rovers $R = \{r_1, \ldots, r_n\}$ and a map of k locations $L = \{l_1, \ldots, l_k\}$. A state in that domain is modelled using the set of state variables $X = \{C_r, I_r, D_{r,l}, B_l, W \mid r \in R, l \in L\}$, where:

- $C_r \in L$ specifies the current location of rover $r \in R$;
- I_r is a Boolean variable which indicates whether the sensor of rover r is initialized;
- $D_{r,l}$ is a Boolean variable which specifies if the rover r has acquired data from location l;
- B_l is a Boolean variable which indicates whether the base station has received the data of location l; and
- W is a Boolean variable which indicates if the wireless channel is available.

2.2.2 Time Uncertainty

A *time random variable* $t \in T$ marks the occurrence of an event, corresponding to either the start or the end of an action. The time random variable $t_0 \in T$ is reserved for the initial time. Each action a has a duration represented by a random variable d_a. A time random variable $t \in T$ is defined by an equation specifying the time at which the associated event occurs.

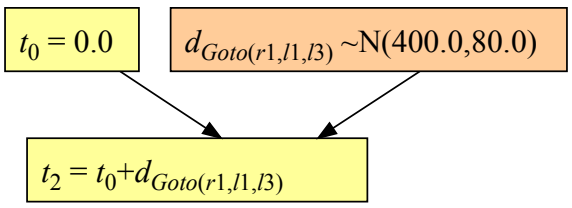


Figure 2.2: Example of a Bayesian network to model time uncertainty

The time random variables are organized in a Bayesian network. For instance, the duration of the $Goto(r_1, l_1, l_3)$ action in the Rovers domain can be modelled using a continuous random variable $d_{Goto(r_1,l_1,l_3)}$ following a normal distribution. Figure 2.2

illustrates a sample Bayesian network representing the end time t_2 which corresponds to the end of a the $Goto(r_1, l_1, l_3)$ started at time t_0. As explained in Section 2.2.5, the duration of the *Goto* action follows a normal distribution having two parameters: a mean μ and a standard deviation σ.

2.2.3 Determined States

Planning with action concurrency requires dealing with undetermined states because the actual outcomes of actions cannot be known before their end. Before introducing undetermined states, the notion of a determined state is defined [2]. Basically, a *determined state* is a set of assigned state variables which are associated to time random variables [9]. Formally, a *determined state* s is defined by $s = (\mathcal{U}, \mathcal{V}, \mathcal{R})$ where:

- $\mathcal{U}$ is a total mapping function $\mathcal{U} : X \rightarrow \cup_{x \in X} Dom(x)$ which retrieves the *current assigned value* for each state variable $x \in X$ such that $\mathcal{U}(x) \in Dom(x)$;
- $\mathcal{V}$ is a total mapping function $\mathcal{V} : X \rightarrow T$ which denotes the *valid time* at which the assignation of variables X have become effective; and
- $\mathcal{R}$ is a total mapping function $\mathcal{R} : X \rightarrow T$ which indicates the *release time* on state variables X.

The release times of state variables track conditions of actions that must be maintained over all the duration of the actions. The time random variable $t = \mathcal{R}(x)$ means that a change of the state variable x cannot be initiated before time t. Valid time ($\mathcal{V}$) and the release time ($\mathcal{R}$) are similar to the write-time and the read-time in Multiple-Timelines of SHOP2's [52], except that we are dealing with random variables instead of numerical values.

Figure 2.3 illustrates an example of two determined states. Because space is limited, only relevant state variables are shown. State s_1 represents two rovers r_1 and r_2 located at locations l_1 and l_2. The sensors of both rovers are initialized (T stands for true) but at different times t_1 and t_0. No data has been acquired or transmitted yet (F stands for false). State s_2 is explained in Section 2.2.6.

2. The definition of determined states is based on the definition of states of ActuPlan as presented in Section 1.2.3

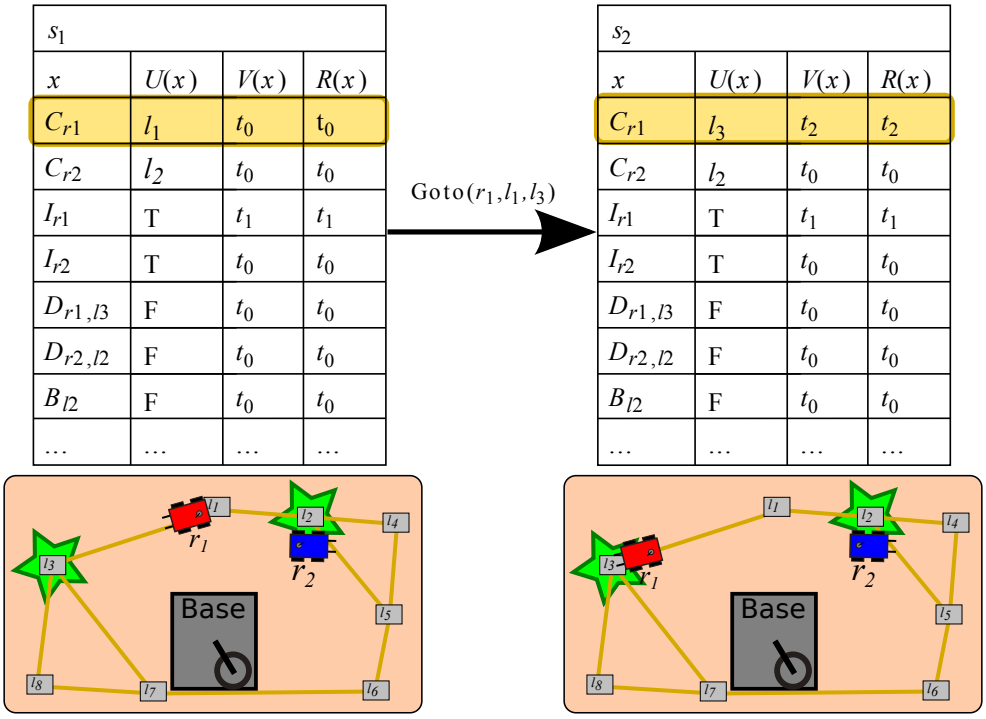


Figure 2.3: Example of determined states

2.2.4 Actions

The specification of actions follows the extensions introduced in PDDL 2.1 [31] for expressing temporal planning domains. The set of all actions is denoted by A. Roughly, an *action* $a \in A$ is a tuple $a = (name, cstart, coverall, estart, O, c_a, d_a)$, where:

- *name* is the name of the action;
- *cstart* is the set of *at start* conditions that must be satisfied at the beginning of the action;
- *coverall* is the set of persistence conditions that must be satisfied *over all* the duration of the action;
- *estart* is respectively the sets of *at start* effects on the state variables;
- O is a set of all possible outcomes occurring at the end of the action;
- and $c_a \in \mathcal{C}$ is the random variable which models the cost of the action;
- and $d_a \in D$ is the random variable which models the duration of the action.

A condition c is a Boolean expression over state variables. The function $vars(c) \rightarrow 2^X$ returns the set of all state variables that are referenced by the condition c. An object effect $e = (x, exp)$ specifies the assignation of the value resulting from the evaluation of expression exp on the state variable x. Expression $conds(a)$ returns all conditions of action a.

An outcome $o \in O$ is a tuple $o = (p, eend)$ where p is the probability that the action has o as outcome, $eend$ are *at end* effects. An effect $e = (x, exp)$ specifies the assignation of the value resulting from the evaluation of expression exp to the state variable x. The set of outcomes O is composed of mutual exclusive outcomes. The sum of the probability of all outcomes of an action must be equals to 1. The expression $effects(o)$ returns all effects of outcome o and $effects(a)$ returns all effects of all outcomes of action a.

To simplify the explanations provided in this paper, the duration is associated to the action itself. This model can be easily generalized by associating a specific random variable for each particular outcome of each action.

An action a is applicable in a state s if and only if s satisfies all *at start* and *over all* conditions of a. A condition $c \in conds(a)$ is satisfied in state s if c is satisfied by the current assigned values of state variables of s.

The application of an action to a determined state updates the functions $\mathcal{U}, \mathcal{V}, \mathcal{R}$ [9]. Following the AI planning PDDL action specification language, actions have *at start* conditions enforcing preconditions at the start of an action. An action starts as soon as all its *at start* conditions are satisfied.

2.2.5 Actions in the Mars Rovers Domain

The specification of actions for the simplified Mars Rovers domain is given in Table 2.1. The action $Goto(r, l_a, l_b)$ describes the movement of the rover r from location l_a to location l_b. The duration of a *Goto* action is modelled by a normal distribution where both the mean and the standard deviation are proportional to the distance to travel. The action $AcquireData(r, l)$ represents the acquisition of data by the rover r in location l using its specialized sensor. Before each data acquisition, the sensor has to be initialized. The action *AcquireData* is nondeterministic and has two

Table 2.1: Actions specification for the Rovers domain

$Goto(r \in R, a \in L, b \in B)$	
cstart	$C_r = a$
eend	$C_r = b$
duration	$Normal(distance(a,b)/speed, 0.2 * distance(a,b)/speed)$
$AcquireData(r \in R, l \in L)$	
coverall	$I_r = true$
coverall	$C_c = l$
eend	$I_r = false$
eend	$D_{r,l} = [0.7]true$ or $[0.3]false$
duration	$Uniform(30, 60)$
$InitSensor(r \in R)$	
cstart	$I_i = false$
eend	$I_r = true$
duration	$Uniform(30, 60)$
$TransmitData(r \in R, l \in L)$	
cstart	$D_{r,l} = true$
cstart	$w = false$
estart	$w = true$
eend	$w = false$
eend	$B_l = false$
duration	$Normal(400, 100)$

possible outcomes: it may succeed with a probability of 0.7 or fail with a probability of 0.3. To simplify the notation, only the nondeterministic effect is specified with probabilities. The initialization of the sensor of a rover r is carried out by the action $InitSensor(r)$. Once data is acquired, it has to be transmitted to the base station using a wireless link. The action $TransmitData(r, l)$ corresponds to the transmission of data acquired from location l by the rover r to the base station. The wireless link has a limited capacity; only one rover can send data at any given time. A rover may transmit data while it is navigating, initializing its sensor or acquiring new data. It is assumed that rovers have an unlimited capacity of storage.

Algorithm 8 Apply action function

```
1.  function Apply(s, a)
2.     s' ← s
3.     t_conds ← max_{x∈vars(conds(a))} s.V(x)
4.     t_release ← max_{x∈vars(effects(a))} s.R(x)
5.     t_start ← max(t_conds, t_release)
6.     t_end ← t_start + d_a
7.     for each c ∈ a.coverall
8.        for each x ∈ vars(c)
9.           s'.R(x) ← max(s'.R(x), t_end)
10.    for each e ∈ a.estart
11.       s'.U(e.x) ← eval(e.exp)
12.       s'.V(e.x) ← t_start
13.       s'.R(e.x) ← t_start
14.    for each e ∈ a.eend
15.       s'.U(e.x) ← eval(e.exp)
16.       s'.V(e.x) ← t_end
17.       s'.R(e.x) ← t_end
18.    for each e ∈ a.enum
19.       s'.W(e.y) ← eval(e.exp)
20.    returns s'
```

2.2.6 State Transition

The application of a deterministic action in a state causes a state transition [3]. Algorithm 8 describes the Apply function which computes the determined state resulting from application of an action a to a determined state s. Time random variables are added to the Bayesian network when new states are generated. The start time of an action is defined as the earliest time at which its requirements are satisfied in the current state. Line 3 calculates the time t_{conds} which is the earliest time at which all *at start* and *over all* conditions are satisfied. This time corresponds to the maximum of all time random variables associated to the state variables referenced in the action's conditions. Line 4 calculates time $t_{release}$ which is the earliest time at which all persistence conditions are released on all state variables modified by an effect. Then at Line 5, the time random variable t_{start} is generated. Its defining equation is the max of all time random variables collected in Lines 3–4. Line 6 generates the time random variable t_{end} with the equation $t_{end} = t_{start} + d_a$. Once

3. In this thesis, Section 2.2.6 is adapted from Section 1.2.6. Algorithms 1 and 8 are identical.

generated, the time random variables t_{start} and t_{end} are added to the Bayesian network if they do not already exist. Lines 7–9 set the release time to t_{end} for each state variable involved in an *over all* condition. Lines 10–17 process *at start* and *at end* effects. For each effect on a state variable, they assign this state variable a new value, set the valid and release times to t_{start} and add t_{end}. Line 18–19 process numerical effects.

Figure 2.3 illustrates the result (State s_2) of applying *Goto(*r_1*,* l_1*,* l_3*)* action to s_1. Because $C_{r_1} = l_1$ is the condition for starting the action, its start time is the release time $\mathcal{R}(C_{r_1}) = t_0$. The action finishes at time $t_2 = t_0 + d_{Goto(r1,l1,l3)}$. Accordingly, the valid ($\mathcal{V}$) and release ($\mathcal{R}$) times of the state variable C_{r_1} are set to t_2.

2.2.7 Quantum States

To plan concurrent actions under uncertainty, a planner has to make decisions (starting actions) even when the outcomes of other running actions are still undetermined. In order to efficiently model such decisions, *planning quantum states* are introduced. The terminology of planning quantum state is a inspired from quantum physic theory where a quantum state is a superposition of classical states. Roughly, in our approach, the start of an action generates a quantum state which is the superposition of all possible future determined states.

A *planning quantum state* is reminiscent of a belief state as generally understood in uncertainty reasoning. A belief state generally represents uncertainty (of an agent) about the current world state which is partially observable. However, even when the current state is unknown, it is always completely determined in the reality. In the presence of concurrency and uncertainty, the current state can be really undetermined because some nondeterministic actions might still running. Thus, a part the current state cannot be observed, not because sensors are imperfect, but the future cannot be observed. In our approach, states are reputed to be undetermined until they need to be observed. Indeed, a real superposition of future possible states is assumed.

Given that a quantum state may contain determined and undetermined state variables, the notion of a *partial state* is required. A *partial state* is a partial assignment of state variables; that is, it assigns values to a subset $\breve{X} \subseteq X$ of the state variables. Formally, a *determined partial state* is defined as $\breve{s}_{\breve{X}} = (\breve{\mathcal{U}}, \breve{\mathcal{V}}, \breve{\mathcal{R}})$, where $\breve{\mathcal{U}}$, $\breve{\mathcal{V}}$ and $\breve{\mathcal{R}}$

have the same meaning as $\mathcal{U}$, $\mathcal{V}$ and $\mathcal{R}$ in the definition of a *determined state* (see Section 2.2.3), except that they are partial mapping functions on a nonempty subset of the state variables $\breve{X} \subseteq X$.

A *superposition of determined partial states* $\overline{s}_{\breve{X}}$ is a set of *determined partial states* where each partial state $\breve{s}_{\breve{X}} \in \overline{s}_{\breve{X}}$ is defined on the same set of state variables $\breve{X} \subseteq X$. The line over $\overline{s}$ denotes a superposition of determined partial states. In a superposition of determined partial states $\overline{s}_{\breve{X}}$, each partial state $\breve{s}_{\breve{X}}$ has a probability of $P(\breve{s}_{\breve{X}}|\overline{s}_{\breve{X}})$ to be observed.

A *quantum state* is a set of superpositions of determined partial states. Formally, a *quantum state* q is defined by $q = \left\{\overline{s}_{\breve{X}_1}, \ldots, \overline{s}_{\breve{X}_n} \mid \cup_i \breve{X}_i = X \ \wedge \ \breve{X}_i \cap \breve{X}_j = \emptyset,\ i \neq j\right\}$. The subsets $\breve{X}_i$ constitute a partition of X. The superpositions composing a quantum state are always independent to each other. The set of the quantum state space is denoted by Q.

A superposition of determined partial states $\overline{s}$ is *completely determined* when $|\overline{s}| = 1$, i.e. the superposition contains exactly one determined partial state. A state variable $x \in \breve{X}$ is *completely determined* when its corresponding superposition of partial states $\overline{s}_{\breve{X}}$ is also completely determined. A quantum state is completely determined when all its superpositions of partial states are completely determined. When a quantum state q is completely determined, it can be projected on a completely determined state $s \in S$.

Figure 2.4 presents two examples of quantum states. The quantum state q_2 is defined by the state s_2 of Figure 2.3 where both rovers has their sensor initialized. The quantum state q_3 is obtained by applying the action *AcquireData*(r_1, l_3) from q_2. Given that $C_{r_1} = l_3$ and $I_{r_1} = \mathrm{T}$ are conditions, the action start time t_3 is defined by the maximum of involved valid times, i.e. $t_3 = max(t_1, t_2)$ The end time is $t_4 = t_3 + d_{AcquireData(r_1,l_3)}$. The resulting quantum state q_3 is a composition of two superpositions of partial states. The first, $\overline{s}_{3,0}$, is completely determined and contains the unchanged state variables. Because the action has a probabilistic outcome, the second superposition $\overline{s}_{3,1}$ is composed of two determined partial states. The first line of a superposition shows the probability of each partial state to be observed. In the first column of a superposition, state variables x and their valid and release times are formatted with a compact syntax $x@\mathcal{V}(x)/\mathcal{R}(x)$. The remaining columns present the

values of $\mathcal{U}(x)$ corresponding to each superposed partial state. Since $C_{r_1} = l_3$ must be maintained over all the duration of the action, its release time $\mathcal{R}(C_{r_1})$ is set to the ending time t_4. The superposition of partial states $s_{3,1}$ represents the two possible outcomes on the D_{r_1,l_1} state variable. Similarly, the quantum state q_4 is obtained by applying the action $AcquireData(r_2, l_2)$ and contains three superpositions of partial states.

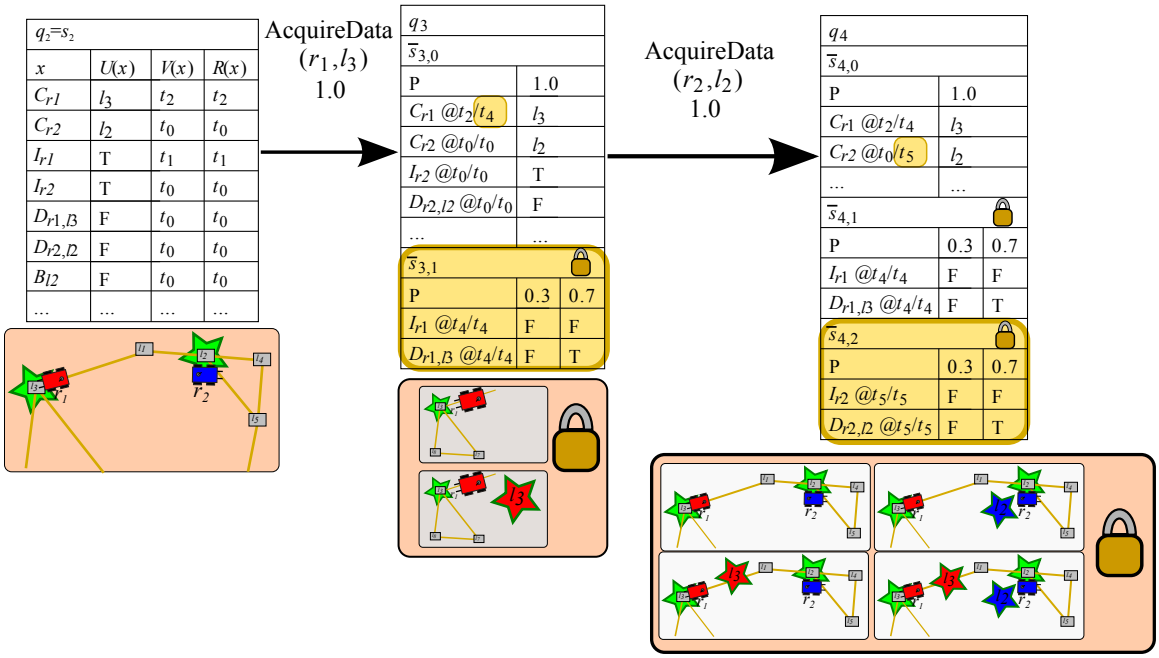


Figure 2.4: Example of quantum states

2.2.8 Observation of State Variables

As in quantum physics, a quantum state is only determined (collapsed) when it is observed [41]. An observation causes a determination which reduces the concerned superposition $\overline{s}_{\breve{X}}$ to a particular determined partial state $\breve{s}_{\breve{X}} \in \overline{s}_{\breve{X}}$. The observation of state variables is required because actions are not applicable when their conditions or outcomes involve undetermined state variables. The observation of a state variable

x is represented as a special action $ActionObserve(x)$ in QuanPlan.

Algorithm 9 Observation of a state variable in a quantum state

1. function Observe($x \in X,\ q = \{\overline{s}_{\breve{X}_1}, \ldots, \overline{s}_{\breve{X}_n}\}$)
2. $\quad i \leftarrow$ find i such that $x \in \breve{X}_i$
3. $\quad Q_r \leftarrow \emptyset$
4. $\quad$ for each $\breve{s}_{\breve{X}_i} \in \overline{s}_{\breve{X}_i}$
5. $\quad\quad q' \leftarrow q \setminus \overline{s}_{\breve{X}_i}$
6. $\quad\quad t_{determ} \leftarrow \max_{x' \in \breve{X}_i} \breve{s}_{\breve{X}_i}.\mathcal{V}(x')$
7. $\quad\quad \breve{s}'_{\breve{X}_i} \leftarrow \breve{s}_{\breve{X}_i}$
8. $\quad\quad \overline{s'}_{\breve{X}_i} \leftarrow \{\breve{s}'_{\breve{X}_i}\}$
9. $\quad\quad P(\breve{s}'_{\breve{X}_i} \mid \overline{s'}_{\breve{X}_i}) \leftarrow 1.0$
10. $\quad\quad q' \leftarrow q' \cup \overline{s'}_{\breve{X}_i}$
11. $\quad\quad$ for each $\breve{X} \in q'$
12. $\quad\quad\quad$ for each $x' \in \breve{X}$
13. $\quad\quad\quad\quad \breve{s}'_{\breve{X}_i}.\mathcal{V}(x') \leftarrow \max(\breve{s}'_{\breve{X}_i}.\mathcal{V}(x'), t_{determ})$
14. $\quad\quad P(q' \mid ActionObserve(x), q) \leftarrow P(\breve{s}_{\breve{X}_i} | \overline{s}_{\breve{X}_i})$
15. $\quad\quad Q_r \leftarrow Q_r \cup q'$
16. $\quad$ return Q_r

Algorithm 9 describes the *Observe* function which is the core of the special action *ActionObserve*. It returns the set of all possible quantum states when observing a state variable x in a quantum state q. Line 2 selects the i^{th} superposition of partial states in q such that $x \in \breve{X}_i$. The observation of state variable x causes the simultaneous determination of all state variables in $\breve{X}_i$. The loop starting at Line 4 iterates on all partial states in the concerned superposition of the state variable x. Each partial state $\breve{s}_{\breve{X}_i}$ corresponds to a specific outcome of a started nondeterministic action. Each iteration generates a quantum state q', which is initialized at Line 5 by removing the superposition of partial states $\overline{s}_{\breve{X}_i}$. Line 6 computes the determination time t_{determ}. It corresponds to the maximum of valid times of involved time random variables. Line 7 copies the current partial determined state to $\breve{s}'_{\breve{X}_i}$. Line 8 constructs a completely determined partial state superposition $\overline{s'}_{\breve{X}_i}$ which only contains the determined partial state $\breve{s}'_{\breve{X}_i}$. Because it will be completely determined in the resulting quantum state q', it has a probability of 1 (Line 9). The observation of a state variable causes the time to advance to time t_{determ}. Line 10 adds the superposition related to state variable x to the quantum state q'. Lines 11–13 force the valid time associated to all state variables to be at least time t_{determ}. This step is important to make sure

that no action can be started before time t_{determ} in quantum state from q' and its descendents. Line 14 sets the probability of quantum state transition for the special action of observing x. Line 15 adds the constructed quantum state q' to the set of resulting quantum states Q_r, which is returned at Line 16.

Figure 2.5 illustrates an example of observing the state variable D_{r_1,l_3} in the quantum state q_3. The determination generates two quantum states q_4 and q_5. The probabilities of observation in the original superposition correspond to the probability of transitions.

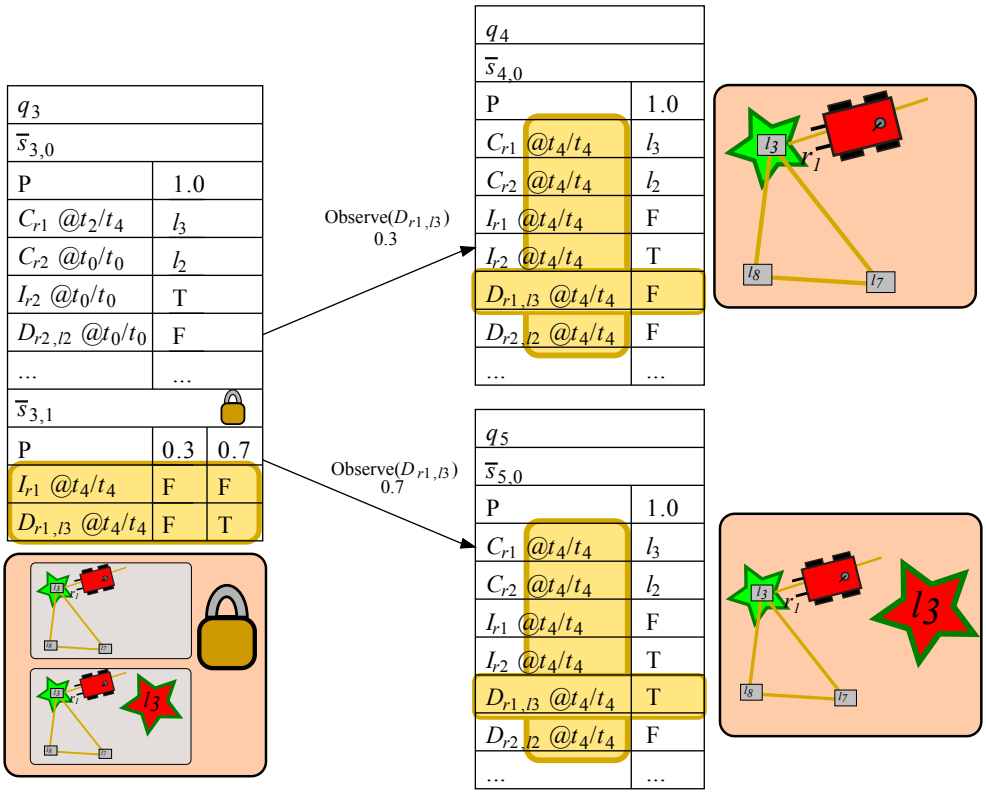


Figure 2.5: Example of an observation of a state variable

2.2.9 Goal

A *goal* $\mathcal{G} = \{g_1, \ldots, g_n\}$ is a set of n timed goal state features. A timed goal state feature $g = (x, v, t) \in \mathcal{G}$ means that state variable x has to be assigned the value v

within $t \in \mathbb{R}^+$ time. A goal $\mathcal{G}$ is satisfied in a quantum state q (denoted $q \models \mathcal{G}$) when the relevant states variables are completely determined to the desired values. Note that the satisfaction of a goal ($q \models \mathcal{G}$) is implicitly a Boolean random variable.

2.3 Policy Generation in the Quantum State Space

QuanPlan applies the dynamic programming of MDP [12] to generate plans. A planning problem is an MDP defined by a 7-tuple $(Q, \mathcal{A}, C, Pr, s_0, \mathcal{G})$, where:

- Q is a set of quantum states which is finite when horizon time is bounded.
- $\mathcal{A} = A \cup \{ActionObserve(x) \mid x \in X\}$ is an augmented set of actions where A are the actions of the planning domain and the actions $ActionObserve(x)$ such $x \in X$ are special actions of QuanPlan.
- $C = \{C_a \mid a \in A\}$ is an optional set of random variables representing the cost of actions.
- $P(q' \mid a, q)$ is the probabilistic transition model which returns the probability of reaching the quantum state $q' \in Q$ by applying the action $a \in \mathcal{A}$ from $q \in Q$. Note that transitions on actions in A are deterministic. The transition probabilities of *ActionObserve* actions are computed by the Line 14 of Algorithm 9.
- $q_0 \in Q$ is the initial quantum state (generally completely determined).
- $\mathcal{G}$ is the goal to satisfy.

The Bellman value function $J : Q \to \mathbb{R}$ is defined by Equation (2.1). $Q_{q,a}$ is used as a shorthand for the set of resulting quantum states when applying action $a \in \mathcal{A}$ from quantum state q. The value $J(q)$ of a quantum state $q \in Q$ represents the cost of the total execution of a plan which passes by quantum state q. The evaluation of values implies the estimation of random variables in the Bayesian network, which is done by a direct sampling algorithm [9]. When q satisfies the goal $\mathcal{G}$, noted $q \models G$, the value $J(q)$ equals the expected value of the maximum of all time random variables multiplied by a weight factor α. The α parameter is used to mix the expected makespan of a plan to the expected cost of actions. Otherwise, if $q \not\models \mathcal{G}$, the value function $J(q)$ takes as value the lowest cost of an action cost plus the weighted value of successor states.

$$J(q) = \begin{cases} \alpha \times E\left[\max_{x \in X} q.\mathcal{V}(x)\right] & \text{if } q \models G \\ \min_{a \in A}\left(E[c_a] + \sum_{q'} P(q'|a,q)J(q')\right) & \text{otherwise} \end{cases} \tag{2.1}$$

A solution plan is a policy π defined by mapping function $\pi : Q \rightarrow A$. A policy returns an action $a \in \mathcal{A}$ for each quantum state $q \in Q$. A policy is given by Equation (2.2).

$$\pi(q) = \arg\min_{a \in \mathcal{A}}\left(E[c_a] + \sum_{q'} P(q'|a,q)J(q')\right) \tag{2.2}$$

2.3.1 Example of Partial Search in the Quantum State Space

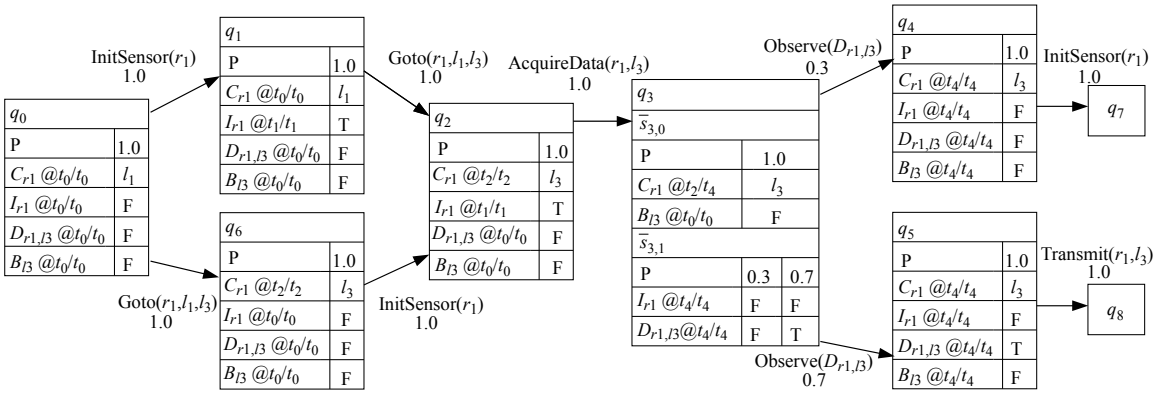


Figure 2.6: Example of expanded quantum state space

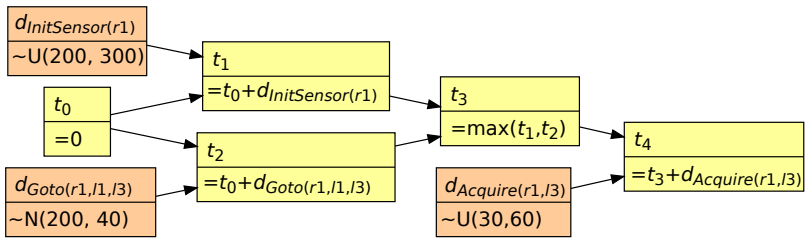


Figure 2.7: Example of expanded Bayesian network

Figure 2.6 presents a partial search graph for a Rovers problem where the goal $\mathcal{G}$ is

to acquire and transmit a data from location l_3. The Bayesian network in Figure 2.7 illustrates the equations of time random variables and the probability distributions followed by the action duration random variables. Only relevant states variables are shown to save space. Quantum state q_0 represents the initial state where the rover r_1 is initially located at location l_1, its sensor is not initialized and no data has been acquired or transmitted yet. The quantum state q_1 is obtained by applying action $InitSensor(r_1)$ from state q_0. The quantum states q_2, q_3, q_4, q_5 are the same as previous examples except that state variables related to rover r_2 are not shown. The quantum state q_6 is obtained by first starting the action $Goto(r_1, l_1, l_3)$ from q_0. Applying the action $InitSensor(r_1)$ from q_6 leads to q_2. The quantum state q_4 represents the observation of a failure of data acquisition. A contingency branch starts with the reinitialization of the sensor (q_7). The quantum state q_5 represents a successful data acquisition. Applying action *Transmit* from q_5 leads to q_8 which satisfies the goal $\mathcal{G}$. Random variables representing durations in Figure 2.7 follow normal (N) and uniform (U) distributions.

2.3.2 Advanced Policy Generation

In the previous formulation, policies map exactly one action to be started for each state. Adopting these policies does not guarantee an optimal behaviour because the observed durations of actions at execution are not considered in the decision making process.

To be optimal, a decision policy must consider the current time λ in addition to the current state q. A decision is to select an action to be started. However, the best action to be started may be not ready because it requires the completion of other actions. At execution, the time random variables in the Bayesian network implicitly become evidences as soon their value can be observed, i.e. when related actions are completed. An action a is *ready* in current state q at current time λ if and only if $q.\mathcal{V}(x) \leq \lambda$ for all involved state variables x in the conditions of a. The decision is made by conditioning Bellman values $J(q)$ on the start time of actions ($J(q)$ is now a random variable). Starting a ready action a at time $t_a = \lambda$ has $E[J(Q_{q,a}) \mid t_a = \lambda]$ as value. If an action is not ready, i.e. it requires the completion of other actions, its

start time t_a is unknown, but has $t_a > \lambda$ constraint. Thus, the value of waiting for starting a not ready action a is $E[J(Q_{q,a}) \mid t_a > \lambda]$.

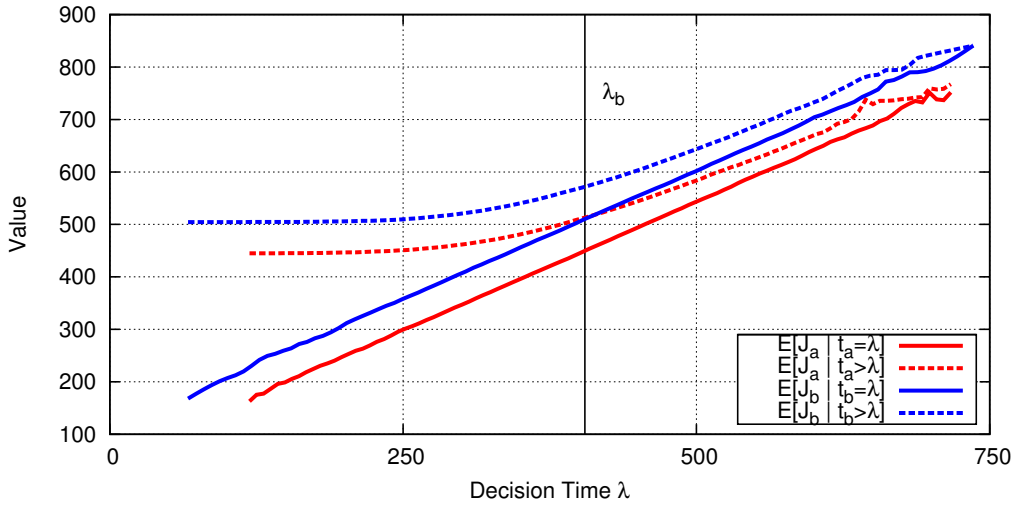


Figure 2.8: Values of actions in a state q

The Figure 2.8 illustrates how a decision is made into a quantum state q at current time λ, where two actions a and b are applicable. If it is ready, the action a is the best action to be started even b is also ready because a has to lowest cost. If a is not ready but b is ready, starting action b is the best choice until a critical time $\lambda_{q,b} = 400$ at which it becomes preferable to wait for a instead starting b.

QuanPlan expands a subset of the quantum state space $Q_e \in Q$ using the Labelled Real Time Dynamic Programming (LRTDP) [16] algorithm. Then for each expended state $q \in Q_e$, a critical time $\lambda_{q,a}$ is computed for each action $a \in \mathcal{A}$. The structure of a policy π is a mapping function $\pi : Q \to \mathbb{R}^n$ where $n = |\mathcal{A}|$. At execution, the first action which becomes ready before its critical time is selected in each state.

2.3.3 Optimality

QuanPlan provides guaranties about optimality and completeness. Because an approximate estimation algorithm is used to estimate the distribution of random variables, the generated policies are also approximately optimal. The distance between

the cost of generated policies and an optimal plan can be bounded to an ϵ under a given confidence level, which is related to the maximum error of estimation errors in the Bayesian network.

2.4 Empirical Results

The Rovers planning domain is used as a benchmark to validate the efficiency of the compact representation of *planning quantum states*. QuanPlan is compared to DUR_{tc}, a concurrent MDP planner based on the works of Mausam and Weld (2008), in which we added the support of time constraints (noted $_{\text{tc}}$). To support time constraints, the interwoven state representation is augmented by a current time variable λ. Both planners implement the LRTDP [16] algorithm and use equivalent heuristics. The time horizon is also bounded to prevent an infinite state expansion. The FPG planner [18] was also considered because it supports action concurrency and uncertainty on duration and outcomes. However, results with FPG were not conclusive because FPG failed to solve most of problems. We suspect that FPG was not able to solve problems because it does not use heuristics that can catch the dependencies between actions.

Table 2.2: Empirical results on the Rovers domains

Problem		QuanPlan		DUR_{tc}		FPG	
$\|R\|$	$\|\mathcal{G}\|$	CPU	Cost	CPU	Cost	CPU	Cost
1	2	0.51	1927	4.99	1932	4	2340
1	3	1.33	2553	112	2563	>300	–
1	4	4.04	3001	>300	–	>300	–
1	5	53.3	3642	>300	–	>300	–
2	2	4.1	1305	65.2	1313	>300	–
2	3	14.9	2129	>300	–	>300	–
2	4	56.0	1285	>300	–	>300	–
2	5	>300	–	>300	–	>300	–

Table 2.2 reports results [4] . First two columns report the size of problems, i.e. the

4. Note that reported results in Table 2.2 are made on different problems of those in Table 1.2. Moreover, since problems reported here have uncertainty on the *AcquireData* action, generated plans are not comparable to those of ActuPlan which have no uncertainty.

number of rovers and goals. For each planner, the running CPU time (seconds) and the cost (expected makespan) of generated plans are reported. No cost are reported for unsolved problems within the maximum allowed planning time (300 seconds). These results demonstrate that our approach performs better than the discrete time planner DUR_{tc}. For some problems, the results of QuanPlan and DUR_{tc} are slightly different. Two reasons explain that. Because QuanPlan relies on a direct sampling algorithm to estimate the random variables, an estimation error (typically ± 5 units) is possible. Second reason is that DUR_{tc} is based on a discrete time model. The granularity on time offers a trade-off between the size of the state space and the level of approximation. Here, times are aligned (rounded up) to multiples of 30 units of time.

2.5 Conclusion

This paper presented QuanPlan, a hybrid planner for solving planning problems with concurrent actions having uncertainty on both duration and outcomes. A compact state representation (planning quantum states) has been introduced to compactly represent future possible states based on their undetermined outcomes. To the best of our knowledge, our combination of Bayesian network and MDP is unique. The dynamically generated Bayesian network efficiently model the uncertainty on the duration of action by enabling a continuous time model rather than a discrete one. The MDP framework handles uncertain action outcomes. The experimental results are quite promising and show the efficiency of using our compact state representation. Future works include the investigation of efficient heuristics to scale our approach to larger problems.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds québécois de la recherche sur la nature et les technologies (FQRNT).

Chapitre 3

Application des processus décisionnels markoviens afin d'agrémenter l'adversaire dans un jeu de plateau

Résumé

Les jeux de plateau sont souvent pris en exemple pour l'enseignement d'algorithmes de prise de décisions en intelligence artificielle (IA). Ces algorithmes sont généralement présentés avec beaucoup d'accent sur la maximisation du gain pour le joueur artificiel. D'autres aspects, comme l'expérience de jeu des joueurs humains, sont souvent négligés. Cet article présente un jeu de plateau simple, basé sur le jeu de serpents et échelles, dans lequel la prise de décisions joue un rôle déterminant en plus de l'incertitude. Ce jeu est utilisé pour introduire des techniques de résolution, des stratégies et des heuristiques afin d'orienter la prise de décision en vue de satisfaire différents objectifs. Un des principaux défis est la génération d'une politique décisionnelle qui procure un défi intéressant à l'adversaire, et ce, tout en s'adaptant automatiquement à son niveau. Les solutions présentées sont basées sur les processus décisionnels de Markov (MDP).

Commentaires

Cet article a été publié et présenté à la conférence *IEEE Computational Intelligence and Games (CIG-2010)* [6]. La contribution de cet article se situe davantage du côté des applications que du côté théorique. L'article vise le domaine des jeux, un créneau très intéressant pour l'application et la mise en valeur de techniques d'intelligence artificielle. Des algorithmes de planification permettent de guider les décisions des personnages artificielles afin d'offrir aux joueurs humains des défis plus stimulants et un meilleur réalisme. L'article s'intègre dans la thèse dans la mesure où le jeu présenté inclut de l'incertitude et de la concurrence d'actions. La concurrence d'actions se manifeste dans une situation d'adversité où deux joueurs s'affrontent. Leurs actions alternées sont modélisées à l'aide de macro-actions qui génèrent des transitions dans un MDP.

L'article étant présenté de façon pédagogique, il apporte également une contribution au niveau de l'enseignement des MDP. Certaines idées présentées dans cet article font l'objet d'un travail pratique donné aux étudiants de premier cycle dans le cadre d'un cours d'intelligence artificielle (IFT 615) au Département d'informatique. Éric Beaudry est l'auteur principal de cet article, en plus d'être l'instigateur du projet. Les étudiants à la maitrise Francis Bisson et Simon Chamberland ont participé à la rédaction et à la réalisation des expérimentations, sous la supervision d'Éric Beaudry et de Froduald Kabanza.

Droits de reproduction

Using Markov Decision Theory to Provide a Fair Challenge in a Roll-and-Move Board Game [1]

Éric Beaudry, Francis Bisson, Simon Chamberland, Froduald Kabanza
Département d'informatique, Université de Sherbrooke,
Sherbrooke, Québec, Canada J1K 2R1
eric.beaudry@usherbrooke.ca, francis.bisson@usherbrooke.ca,
simon.chamberland@usherbrooke.ca,
froduald.kabanza@usherbrooke.ca

Abstract

Board games are often taken as examples to teach decision-making algorithms in artificial intelligence (AI). These algorithms are generally presented with a strong focus on winning the game. Unfortunately, a few important aspects, such as the gaming experience of human players, are often missing from the equation. This paper presents a simple board game we use in an introductory course in AI to initiate students to gaming experience issues. The Snakes and Ladders game has been modified to provide different levels of challenges for students. This adaptation offers theoretical, algorithmic and programming challenges. One of the most complex issue is the generation of an optimal policy to provide a fair challenge to an opponent. A solution based on Markov Decision Processes (MDPs) is presented. This approach relies on a simple model of the opponent's playing behaviour.

3.1 Introduction

It is well known that computer science students are often avid video game players. Thus, using games in computer sciences classes is a good teaching strategy to get students interested. It is not surprising that most textbooks [57, 44] in the field of artificial intelligence (AI) use games to introduce formal algorithms. Simple board games such as the n-puzzle, tic-tac-toe, Gomoku and chess are often used in AI courses because they are usually well known by most people. These games also offer the advantage of being easy to implement because they generally have a discrete representation as well as simple rules.

In classical AI courses, the main focus of homework assignments usually is the design of an AI that optimizes its decisions to win the game. Unfortunately, a few important aspects are disregarded and one of them is user experience.

Today, an important trend in the design of video games is to provide a fair challenge to human players. Properly balancing the difficulty level of an intelligent adversary in a video game can prove to be quite a laborious task. The agent must be able to provide an interesting challenge to the human player not to bore him. On the other hand, an opponent that exhibits an optimal behaviour will result in a player that will either be discouraged or accuse his opponent of cheating. Different strategies can be adopted to create an AI with various difficulty levels.

A *rubber band* (i.e., cheating) AI [59] describes an artificial player that is given an advantage over human players through various means. For example, artificial players may have a perfect visibility of the state of the world, obtain better items, or attain greater speeds. Rubber banding is especially common in racing video games (for instance, the Mario Kart series [48]): human players are led to believe they are winning the race, only to see their opponents get speed boosts for dragging behind too much, and zoom right past them at the last moment. A human player experiencing such a situation is likely to be frustrated and stop playing the game.

In this paper, we show how it is possible to use a Markov Decision Process (MDP) [12] solving algorithm to compute a policy for an autonomous intelligent agent that adjusts its difficulty level according to its opponent's skill level. The resulting policy ensures that the artificial opponent plays suboptimally against an inexperienced

player, but also plays optimally when its adversary is leading the game. This allows the opponent to offer a challenge to the human player without exhibiting a cheating behaviour.

A modified version of the well-known Snakes and Ladders board game is used as a testbed and as the game framework for a homework assignment in our introductory AI course for undergraduate students. As opposed to the usual rules of the game where chance totally determines the final state of the game, our modified game allows the players to decide of an action to take at each turn. Three actions are allowed: advancing by one square on the board, throwing one die, or throwing two dice. Since each action has a different probabilistic outcome, the player has to carefully think about which action is the best on each square of the board. The board configuration, i.e., the snakes and ladders, strongly influences the actions to be taken. Since this game is non-deterministic and involves a sequence of decision-making, the Markov Decision Process (MDP) formalism comes as a natural approach to compute optimal policies.

Although the game seems trivial at first glance, it nevertheless offers different types of interesting challenges. The simplest problem in Snakes and Ladders is to decide which actions to take in order to reach the end of the board as quickly as possible. This problem is easily solved using an MDP to compute an optimal policy which assigns an action to each board position. However, in a multiplayer context, adopting this policy is not always the best strategy for winning the game. In many situations, players may have to attempt desperate or riskier moves in order to have a chance to win the game. Consequently, the position of the opponent has to be considered to act optimally. The MDP framework can be used to solve this problem optimally.

A more interesting issue arises when trying to provide a fair challenge to the adversary. One possible solution is to model the gaming experience of the opponent. Instead of generating a policy that exclusively optimizes the winning probability, the MDP could generate a policy which optimizes the opponent's gaming experience.

There are also other types of interesting issues that come with this simple game. For instance, very large virtual boards can be quite hard to solve optimally. Fortunately, many strategies can be used to speed up MDP solving: heuristics to initialize

the values of states, an improved value iteration algorithm like Real-Time Dynamic Programming (RTDP) [5] or Labeled RTDP (LRTDP) [16], and other *ad hoc* programming tricks.

The rest of this paper is organized as follows. Section 3.2 introduces the MDP framework. Sections 3.3 and 3.4 describe, respectively, how to compute an optimal policy to win the game and an optimal policy to optimize user experience. A conclusion follows in Section 3.5.

3.2 Background

Markov Decision Processes (MDPs) are a well-established mathematical framework for solving sequential decision problems with probabilities [12]. They have been adopted in a variety of fields, such as economic sciences, operational research and artificial intelligence. An MDP models a decision-making system where an action has to be taken in each state. Each action may have different probabilistic outcomes which change the system's state. The goal of an MDP solving algorithm is to find a policy that dictates the best action to take in each state. There exist two main formulations for MPDs: one strives to minimize costs and the other aims to maximize rewards.

3.2.1 Minimizing Costs

Some problems, like path-finding, are easier to model using a cost model for the actions. The objective is to compute a policy which minimizes the expected cost to reach a goal state. Formally, an MDP is defined as a 7-tuple $(S, A, P, C, s_0, G, \gamma)$, where:

- S is a finite set of world states;
- A is a finite set of actions that the agent could execute;
- $P : S \times S \times A \to [0, 1]$ is the state probability transition function. $P(s', s, a)$ denotes the probability of reaching state s' when executing action a in state s;
- $C : A \to \mathbb{R}^+$ is the system's cost model;
- $s_0 \in S$ is the initial world state;

– $G \subseteq S$ is the set of goal states to be reached;
– $\gamma \in]0, 1]$ is the discount factor.

A *decision* is the choice to execute an action $a \in A$ in a state $s \in S$. A *policy* is a strategy (a plan), that is, the set of decisions for every state $s \in S$. An *optimal policy* is a policy which assigns the action which minimizes the expected cost to reach the goal in every state.

Several algorithms exist to compute an optimal policy, given a cost model. The value iteration algorithm [12] uses the Bellman equation to compute the best action for each state in a dynamic programming fashion. It starts by computing a value $V(s)$ for each state $s \in S$ by making several iterations of Equation (3.1).

$$V(s) = \begin{cases} 0, \text{if } s \in G \text{ else} \\ \min\limits_{a \in A} \left(C(a) + \gamma \sum\limits_{s' \in S} P(s', s, a) \cdot V(s') \right) \end{cases} \tag{3.1}$$

Once the values of states have converged, an optimal policy can be extracted using Equation (3.2). There may exist several optimal policies since, given a state, it is possible for two or more different actions to have the same minimal expected cost.

$$\pi(s) = \arg\min_{a \in A} \left(C(a) + \gamma \sum_{s' \in S} P(s', s, a) \cdot V(s') \right) \tag{3.2}$$

In other words, each state $s \in S$ is associated with the action $a \in A$ that has the best compromise between cost ($C(a)$) and the expected remaining cost of actions' outcomes. When $\gamma = 1$, this problem is also known as the *stochastic shortest path problem*.

3.2.2 Maximizing Rewards

Other problems are not naturally expressed with a cost model. Consider the robot motion planning domain in Figure 3.1. The map is represented as an occupancy grid where black and grey squares are obstacles, the blue triangle is the robot's initial position and the green circle is the goal. Computing a policy which avoids black and grey squares as much as possible could be done by attributing a positive reward to

the goal state and a negative reward to undesirable states (e.g., obstacles). Thus, the objective with this formulation of MDPs is to compute a policy which maximizes the expected reward in each state.

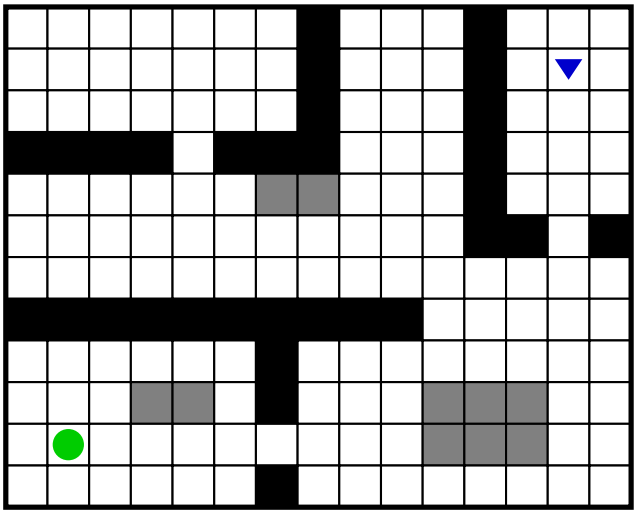


Figure 3.1: Occupancy grid in a robot motion planning domain. Different colours represent the different initial reward values: $Black = -1$, $Grey = -0.4$, $White = 0$ and $Green$ (goal) $= 1$. The robot's initial position is denoted by the blue triangle.

The formal definition of a rewards-maximizing MDP is identical to that of cost-minimizing MDPs, except for the cost model ($C : A \to \mathbb{R}^+$) and the goal G, which are replaced by a rewards model ($R : S \to \mathbb{R}$). This rewards model associates each state with a desirability degree. The Bellman equation for this formulation is given in Equation (3.3).

$$V(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s', s, a) \cdot V(s') \tag{3.3}$$

In this formulation, the best action maximizes the expected reward in every state instead of minimizing the cost to reach the goal. An optimal policy is then extracted using Equation (3.4).

$$\pi(s) = \arg\max_{a \in A} \sum_{s' \in S} P(s', s, a) \cdot V(s') \tag{3.4}$$

3.2.3 Algorithms for Solving MDPs

There exist several algorithms for solving MDP problems, such as value iteration and policy iteration [57]. The value iteration algorithm iteratively evaluates the Bellman equations (Equations 3.1 and 3.3) for each state until they all converge. After convergence, the policy is extracted using Equation (3.2) or Equation (3.4), depending on the chosen MDP formulation.

As its name suggests, the policy iteration algorithm iterates on policies rather than on state values. It starts with an arbitrary policy that is iteratively refined. During an iteration, the max operator in the Bellman equation may be removed, since the policy is fixed; this results in a linear equation system. This linear equation system is solved to compute the state values $V(s_i)$. At the end of each iteration, a new policy is extracted. The optimal policy is obtained when there is no change in two successive iterations.

Some advanced techniques have been proposed for solving MDPs. The Real-Time Dynamic Programming (RTDP) algorithm [5] is a popular technique to rapidly generate good, near-optimal policies. The key idea is that some states have a higher probability than others to be reached during execution. Instead of iterating on the entire state space, the RTDP algorithm begins trials from the initial state, makes a greedy selection over the best action, and then stochastically simulates the successor state according to the state probability transition function. When a goal state is reached, a new trial is started. This process is repeated until convergence of the greedy policy.

To be efficient, the RTDP algorithm requires a heuristic function to initialize the state values. The algorithm is guaranteed to converge to an optimal policy when the heuristic is admissible [16]. The main advantage of the RTDP algorithm is that, when given a good heuristic function, it can produce a near-optimal policy much faster than any value iteration or policy iteration algorithm.

Although RTDP gives good results fast, its convergence is very slow, due to the greedy nature of the selection of states to explore. States with high probabilities of being reached are visited (and their values computed) over and over again, to the detriment of other, less likely states.

Labeled RTDP [16] (or LRTDP for short) is an improved version of RTDP that

consists in labelling states which have already converged to their optimal values. Solved states (i.e., states that have reached convergence) are avoided during the stochastic choice of successor states in the trials, thus allowing the algorithm to visit more states and converge faster towards an optimal policy.

3.3 Optimal Policy for Winning the Game

3.3.1 The Modified Snakes and Ladders Game with Decisions

The Snakes and Ladders game is a roll-and-move game which is played on a grid board. The winner is the player who first reaches the end of the board. In the classic game, players throw two dice and advance their position by the sum of the dice's values. Thus, the game is totally determined by chance. Snakes and ladders linking board squares are spread across the grid. When players reach a square with a ladder, they automatically advance to the square located at the top of the ladder. When players reach a square with a snake's head, they must go back to the square pointed by the tip of the snake's tail. It is also possible for a player to have several successive instantaneous moves (e.g., if a snake starts at the top of a ladder).

The Snakes and Ladders board game has been modified in order to introduce decisions. Each turn, players have to decide of an action from the set of actions $A = \{a_1, a_D, a_T\}$ where:

- a_1 is the action to advance by a single square;
- a_D is to throw one die;
- a_T is to throw two dice.

Each action has a set of possible outcomes which are defined by the function $N : A \to 2^{\{1,\ldots,12\}}$. Outcomes define the number of squares by which the player could advance on the game board. For instance, $N(a_D) = \{1, 2, 3, 4, 5, 6\}$. The probability of outcomes of an action $a \in A$ is denoted by $P(n \in \{1, \ldots, 12\}|a)$. For instance, $P(6|a_T) = \frac{5}{36}$.

Figure 3.2 presents a simple board for the game with $n = 20$ squares. The function $T : \{0, \ldots, n-1\} \times \{1, \ldots, 12\} \to \{0, \ldots, n-1\}$ defines the square the player will be

in after considering the snakes and ladders on the board. For instance in the example board, $T(0,2) = 2$, $T(0,1) = T(0,4) = 4$ and $T(10,1) = 18$. Moves that would bring the player beyond the last board square are prohibited and result in the player not moving. The last position has to be reached with an exact move. Thus, $T(18,1) = 19$ but $T(18,2) = 18$ because position 20 does not exist.

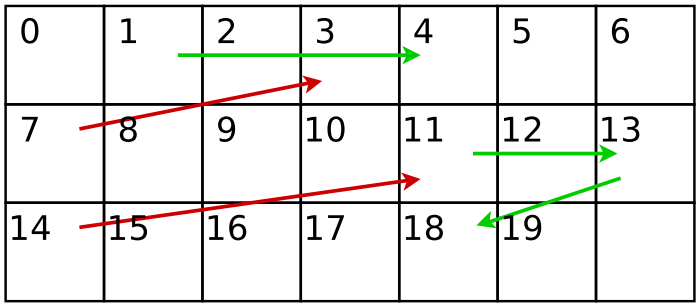


Figure 3.2: Simple board for the Snakes and Ladders game. Red arrows represent "Snakes", and green arrows represent "Ladders".

3.3.2 Single Player

The simplest problem in the modified game of Snakes and Ladders is to decide which actions to take in order to reach the end of the board as quickly as possible without considering the opponent. This problem is easily solved using an MDP to compute an optimal policy which attributes an action to each board position. Since the goal is to minimize the expected number of turns, the cost formulation of MDPs is the most appropriate one. The state space S is defined as the set of states for each board position $S = \{s_0, s_1, ..., s_{n-1}\}$, where s_0 is the initial state and $s_{n-1} \in G$ is the final (goal) state. The probabilistic state transition model is defined by Equation (3.5).

$$P(s_j, s_i, a) = \sum_{x \in N(a) | T(i,x)=j} P(x|a) \tag{3.5}$$

Since the state horizon is finite (the game ends once the last square is reached) and the goal is to find the shortest stochastic path, the discount factor $\gamma = 1$. All

actions cost one turn; thus, the $C(a)$ term could simply be replaced by 1 and removed from the summation. By merging Equations 3.1 and 3.5, we obtain Equation (3.6).

$$V(s_i) = \begin{cases} 0, \text{if } i = n-1 \text{ else} \\ 1 + \min\limits_{a \in A} \sum\limits_{x \in N(a)} P(x|a) \cdot V(s_{T(i,x)}) \end{cases} \tag{3.6}$$

The value iteration algorithm can be implemented in a straightforward manner simply by programming Equation (3.6). For small boards, the policy generation is very fast. However, on larger board (thousands or millions of states) the convergence could be very slow if implemented in a naïve way. To speed up convergence, many strategies can be used.

The value iteration algorithm updates the state values V during several iterations until convergence. Most AI textbooks present this algorithm as the process of updating a V vector using the values V' from the previous iteration. A faster implementation may be achieved by updating a unique vector of V values instead. Updated values are thus used sooner.

Another strategy that could be added to this one is the use of a particular ordering for iterating on states. Iterating in the state order is very slow because several iterations is necessary before cost values propagate from the goal state to the initial state. Thus, starting each iteration from the final state results in much faster convergence.

Table 3.1 shows empirical results for a few iterations of the algorithm on the board from Figure 3.2. The first column indicates the states. The next columns presents the current $V(s_i)$ value after the n^{th} iteration. After nine iterations, the algorithm has converged and the last column shows the extracted policy. Values in the last iteration column represent the expected number of remaining turns to reach the end of the board.

Figure 3.3 compares the running time of a standard MDP implementation with an optimized one on various board sizes.

Table 3.1: Empirical results for the value iteration algorithm on the board from Figure 3.2

State	Iter#1	Iter#2	Iter#3	...	Iter#9	π
s_0	4.03	3.70	3.67	...	3.66	a_T
s_1	3.75	3.50	3.48	...	3.47	a_T
s_2	3.44	3.28	3.26	...	3.26	a_T
s_3	4.18	3.17	3.08	...	3.07	a_T
s_4	3.67	3.00	2.94	...	2.93	a_T
s_5	3.26	2.91	2.88	...	2.87	a_T
s_6	2.84	2.82	2.81	...	2.80	a_T
s_7	2.82	2.78	2.77	...	2.77	a_T
s_8	2.62	2.61	2.61	...	2.61	a_D
s_9	2.72	2.69	2.67	...	2.67	a_D
s_{10}	2.00	2.00	2.00	...	2.00	a_1
s_{11}	2.42	2.40	2.39	...	2.39	a_T
s_{12}	2.00	2.00	2.00	...	2.00	a_1
s_{13}	2.00	2.00	2.00	...	2.00	a_1
s_{14}	2.00	2.00	2.00	...	2.00	a_T
s_{15}	3.33	3.11	3.04	...	3.00	a_D
s_{16}	3.00	3.00	3.00	...	3.00	a_1
s_{17}	2.00	2.00	2.00	...	2.00	a_1
s_{18}	1.00	1.00	1.00	...	1.00	a_1
s_{19}	0.00	0.00	0.00	...	0.00	a_T

3.3.3 Two Players

Playing with the previous policy which minimizes the number of moves to reach the end of the board as quickly as possible is unfortunately not always the best strategy to win against an opponent. Let us define the state space by considering the position of both players on the board. Consider the board and the generated policy in Figure 3.4. Let the current game state be such that player A is at position 17 and player B is at position 18. What is the best action for player A? Using the optimal single-player policy, A will choose action a_1 and then reach position 18. However, player B will win the game at his next turn. The probability of winning the game in this state is thus zero if we adopt the strategy to reach the end of the board as quickly as possible. In this situation, the best move is a desperate one: by selecting

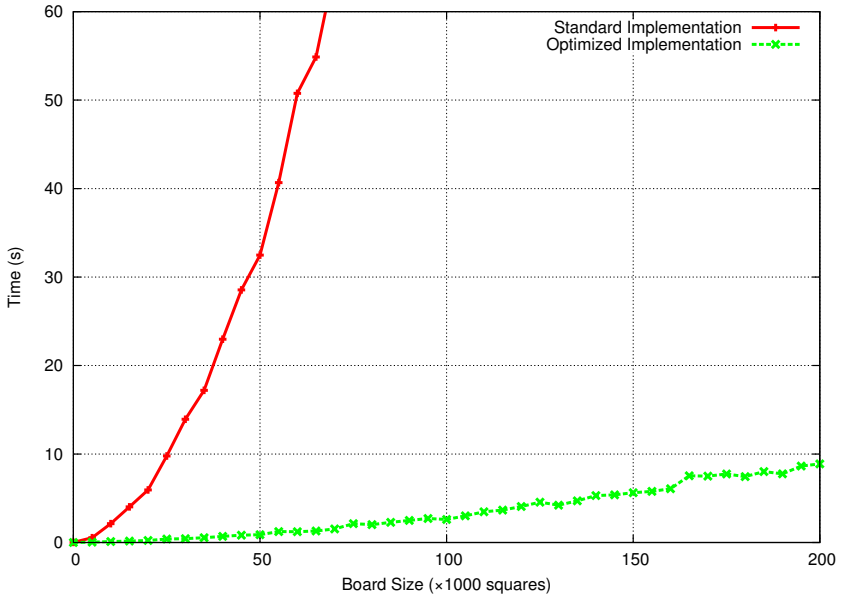


Figure 3.3: Performance improvement of an optimized policy generator

action a_D, player A can hope to move by exactly 2 squares and then win the game. Thus, in this game state, player A still has a probability of $\frac{1}{6}$ to win the game using action a_D.

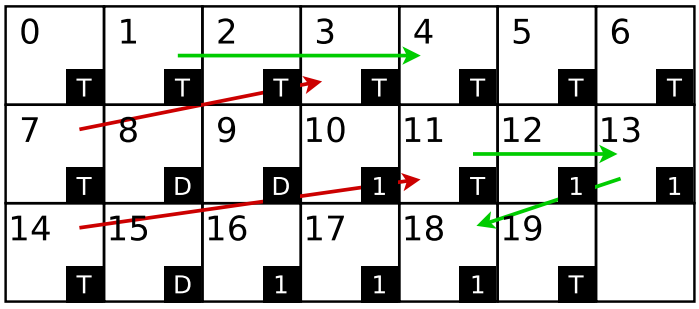


Figure 3.4: Simple board from Figure 3.2 with an optimal single-player policy. Actions a_1, a_D and a_T have been abbreviated to 1, D and T, respectively, for conciseness.

The computation of the best decision to take in a two players context is more challenging than in a single-player context. Indeed, decisions in a multiplayer context do not only depend on the player's position, but also on the opponent's position. A two-players policy associates an action to a pair of positions, and is defined as

$\pi : S \to A$, where $S = \{s_{i,j} \; \forall \; (i,j) \in \{0, \ldots, n-1\}^2\}$.

Provided that the opponent's strategy can be modelled using one such policy π_{sp} (which does not evolve over time), we can calculate a policy $\pi'(\pi_{sp})$ maximizing the chances of winning the game against said opponent. Several algorithms can be used to compute $\pi'(\pi_{sp})$.

Since this is a zero-sum game with two players, one could suggest using Alpha-Beta Pruning-based algorithms. Algorithm 10 presents an adaptation of the classic algorithm to consider chance nodes [3]. A limitation of Alpha-Beta Pruning is that it requires to reach leaf nodes of the search tree to make an optimal decision. Even if a very small board is used, leaf nodes could be very deep in the search tree. Another problem is that the outcome of the actions are probabilistic, which may produce infinite loops with a non-zero probability. A common strategy is to cut the search tree by setting a maximum depth. Nodes at this depth are evaluated using a heuristic function. This evaluation function is generally approximate and cannot guarantee optimality.

To guarantee optimality, an MDP can be used. Since MDPs are designed for sequential decision-making problems, one may question this choice because it does not naturally fit games with adversaries. However, since an assumption is made on the behaviour of the opponent (by using a fixed policy), it is possible to integrate the opponent's choices in the player's decisions.

The cost formulation of MDPs is not exactly appropriate anymore since the goal is not to minimize the expected number of turns, but rather to reach the end of the board before the opponent. We thus adopt the rewards formulation: a reward is simply put on states where the player wins the game. Since all winning states are considered equivalent (winning by a distance of 2 or 200 positions is equivalent) the reward is set uniformly as given by Equation (3.7).

$$R(s_{i,j}) = \begin{cases} 1, \text{if } i = n-1 \wedge j < n-1 \text{ else} \\ 0 \end{cases} \tag{3.7}$$

The transition probability function is defined in such a way as to consider that both players move simultaneously, as described in Equation (3.8).

Algorithm 10 Alpha-Beta Pruning with Chance Nodes

```
1.  AlphaBetaSearch(s_{i,j})
2.    (value, action) ← MaxNodeSearch(s_{i,j}, −∞, +∞)
3.    return action

4.  MaxNodeSearch(s_{i,j}, α, β)
5.    if i = n − 1 return +1
6.    if j = n − 1 return −1
7.    value ← −∞
8.    for each a_i ∈ A
9.      v ← 0
10.     for each x ∈ N(a_i)
11.       i' ← T(i, x)
12.       (v_n, a_j) ← MinNodeSearch(s_{i',j}, α, β)
13.       v ← v + P(x|a_i) · v_n
14.     if v > value
15.       value ← v
16.       action ← a_i
17.     if value ≥ β break
18.     α ← max(α, value)
19.   return (value, action)

20. MinNodeSearch(s_{i,j}, α, β)
21.   if i = n − 1 return +1
22.   if j = n − 1 return −1
23.   value ← +∞
24.   for each a_j ∈ A
25.     v ← 0
26.     for each y ∈ N(a_j)
27.       j' ← T(j, y)
28.       (v_n, a_i) ← MaxNodeSearch(s_{i,j'}, α, β)
29.       v ← v + P(y|a_j) · v_n
30.     if v < value
31.       value ← v
32.       action ← a_j
33.     if value ≤ α break
34.     β ← min(β, value)
35.   return (value, action)
```

$$
\begin{aligned}
M_{i,i'} &= \{x \in N(a) | T(i,x) = i'\} \\
M_{j,j'} &= \{y \in N\left(\pi_{sp}(s_j)\right) | T(j,y) = j'\} \\
P(s_{i',j'}, a, s_{i,j}) &= \sum_{x \in M_{i,i'}} P(x|a) \sum_{y \in M_{j,j'}} P(y|\pi_{sp}(s_j))
\end{aligned}
\tag{3.8}
$$

Integrating Equations 3.3 and 3.8 results in Equation (3.9).

$$
\begin{aligned}
V(s_{i,j}) = \quad & R(s_{i,j}) \\
+ \quad & \max_{a \in A} \sum_{x \in N(a)} P(x|a) \\
& \cdot \sum_{y \in N(\pi_{sp})} P(y|\pi_{sp}) \cdot V(s_{T(i,x),T(j,y)})
\end{aligned}
\tag{3.9}
$$

i\j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	1	T
1	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	1	T
2	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	1	T
3	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	1	T
4	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	1	T
5	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	1	T
6	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	1	T
7	D	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
8	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	T	T
9	1	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	T	T
10	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	T	T
11	1	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
12	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	T	T
13	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	D	T
14	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	D	T
15	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	T
16	1	1	1	1	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	T
17	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	D	T
18	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	T
19	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T

Figure 3.5: Optimal policy to beat an opponent playing with an optimal policy to reach the end of the board as quickly as possible. The row index i gives the position of the AI player and the column index j gives the opponent's position.

Since we consider simultaneous moves in state transition, an important question is what happens when both players reach the end of the board during the same turn. Since we play before the opponent, reaching the state $s_{n-1,n-1}$ from an another state $s_{i,j}$ such that $i, j < n-1$ means that the first player reaches the end before the

opponent. Thus, if a draw is not allowed, a reward should be put on this particular square: $R(s_{n-1,n-1}) = 1$.

Figure 3.5 shows an optimal policy to beat an opponent playing an optimal policy to reach the end of the board as quickly as possible. Note that the optimal policy is not necessarily unique. At each turn, the player in square i, which plays against an opponent in square j, looks up the cell (i, j) to take the best action which maximizes his chance of winning.

Table 3.2 presents results which compare the percentage of games won by the single-player optimal policy (computed as presented in Section 3.3.2) against the two-players optimal policy (computed as presented here) on a board of size $n = 1000$ squares. Results shows an improvement of 3 % of winning chances when using the two-players optimal policy against a single-player optimal one.

Table 3.2: Percentage of wins between single- and two-players policies

vs	Single-Player Policy	Two-Players Policy
Single-Player Policy	50 %	47 %
Two-Players Policy	53 %	50 %

Figure 3.6 presents the required time to generate single-player and two-players policies. Optimal decision-making which considers the opponent comes at a cost: the state space grows quadratically as the board size increases.

3.3.4 Generalization to Multiplayer

Considering more than two players offers a new challenge. Similarly to the single-player to two-players generalization, a simple avenue is to add a new dimension to the state space. Thus, a state could be defined by a m-tuple $s = (p_1, p_2, \ldots, p_m)$ which gives the positions of all m players. A problem with this approach is that the size of the state space grows with the size of the board and the number of players, i.e, $||S|| = n^m$. Solving this problem in an optimal way becomes quickly intractable. A viable approximation for multiplayer is to model the game as a two players game and only consider the opponent that is closest to the goal state.

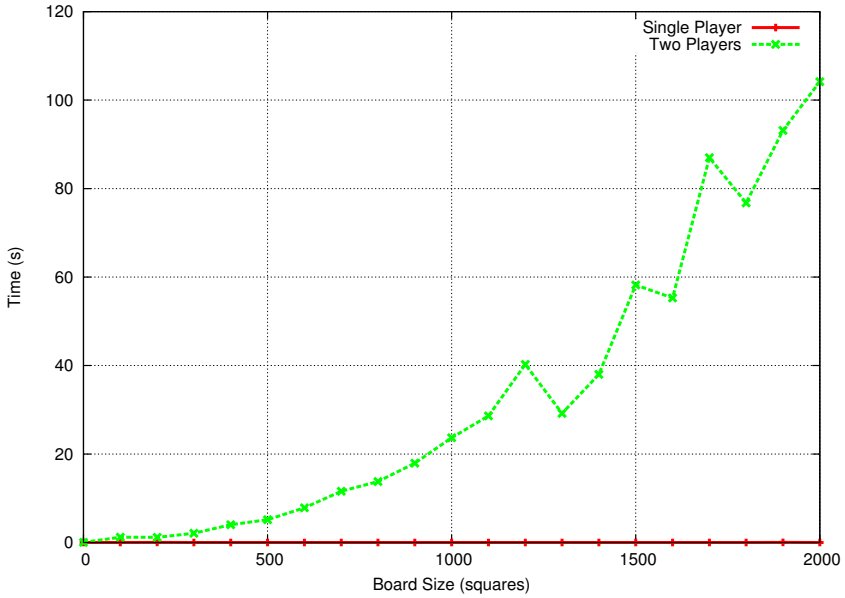


Figure 3.6: Required time to generate single- and two-players policies

3.4 Optimal Policy for Gaming Experience

A more complex aspect of the game involves considering the gaming experience of a human opponent. Playing against an opponent with a similar skill level is generally more fun than playing against someone who is too strong or too weak. Properly balancing the difficulty level of an intelligent adversary in a video game can prove to be quite a laborious task.

Before proposing a solution to this problem, there is an important question we need to answer: what exactly is gaming experience? Without a clear definition, it is difficult to try to maximize it. Thus, a formal gaming experience mathematical model is required. Elaborating such a model is an orthogonal problem to that of optimizing an AI player's decisions to maximize the opponent's gaming experience. Once a gaming experience model is given, it is simply integrated into the equation.

As was the case with the policy developed in Section 3.3.3, an assumption has to be made about the opponent's playing strategy. For the rest of the paper, consider that the opponent does not play optimally but rather only follows his intuition. Because throwing two dice generally allows the player to move closer to the final state, the

opponent player selects an action using the strategy π_{opp} presented by Equation (3.10). Note that the opponent only considers his own position j and does not consider the AI player's position i.

$$\pi_{opp}(s_{i,j}) = \begin{cases} a_T, & \text{if } j < n-6 \\ a_D, & \text{if } n-6 \leq j < n-3 \\ a_1, & \text{if } n-3 \leq j \end{cases} \tag{3.10}$$

3.4.1 Simple Opponent Abandonment Model

As a simple model of gaming experience, an abandonment rule is defined as follows. The opponent player abandons the game if he is too far from the AI player's position on the board, i.e., the opponent believes he has no chance of winning. Another source of abandonment is when the opponent has no challenge, i.e., when the opponent believes that the game is too easy. More precisely, the opponent abandons the game when the distance between the players' positions is greater than half the size of the board.

Thus, the goal of the AI player is to maximize its chance of winning the game before the opponent abandons. This problem can be solved using an MDP in a similar way of the one which maximizes the chance of winning against a given opponent. As done for a two opponents game (Section 3.3.3), i.e., a state is defined as a pair of positions on the board. The main difference is that there is a set of abandonment states $S_{ab} \subset S$ which is defined as $S_{ab} = \{s_{i,j} \; \forall \; (i,j) \in \{0,...,n-1\}^2 : |i-j| \geq \frac{n}{2}\}$, where n is the number of board squares. By definition, states $s \in S_{ab}$ are states where the opponent abandons the game because of the lack of enjoyment (the opponent being too weak or too good). Thus, these states are terminal where no action is applicable. An action applicability function, defined as $App : S \rightarrow A$, is used to find out which actions are applicable in a given state.

Table 3.3 presents empirical results (1 000 000 simulations on a board of size $n = 1000$) that demonstrate how using an optimal policy to win the game results in the abandonment of most games (first column). Instead, a policy computed by taking account of the opponent's model greatly reduces the number of abandonments, while

still exposing an optimal behaviour, in the sense the AI player wins the majority of games. Note that this could also discourage the opponent; the policy could be improved to try to balance the number of wins and losses.

Table 3.3: Improvement when considering the abandonment model

Final Game State	Optimal Policy	Considering Opponent Model
Wins	33.9 %	97.1 %
Losses	0.6 %	1.6 %
Abandonments	65.5 %	1.3 %

Solving MDPs for large state spaces takes very long time to converge to an optimal policy. In most situations, a near-optimal policy, generated with an algorithm such as RTDP, is very much acceptable since these techniques produce good results in a short amount of time. Figure 3.7 empirically compares the performance of value iteration and RTDP on a large game board ($n = 1500$) over 1 000 000 simulations. The quality of the policies, measured in terms of the percentage of victories without the opponent abandoning, is displayed as a function of the allotted planning time. The difference, while not astounding, would still be welcomed in a time-critical context.

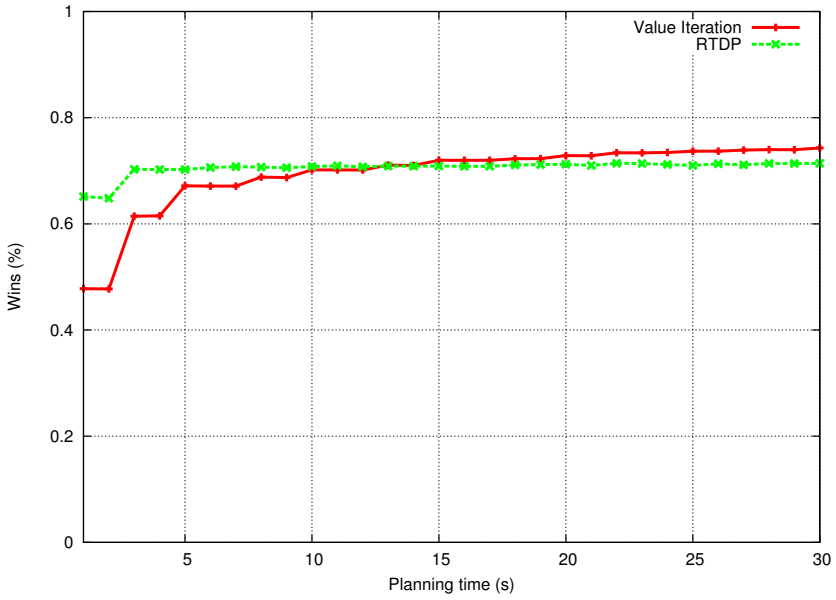


Figure 3.7: Quality of plans as a function of the allotted planning time

This simple abandonment model could be further improved. For instance, the abandonment could be probabilistic instead of deterministic. We could define a probability density function which specifies the probability of abandonment given the position of both players.

3.4.2 Distance-Based Gaming Experience Model

Another user experience model that could be used is a distance-based one. Rather than setting rewards on states which correspond to the end of the game, rewards can be attributed on all states identified by Equation (3.11). The maximum reward (0) is obtained when both players are in the same board position and it decreases quadratically as the distance between both players increases.

$$R(s_{i,j}) = -(i-j)^2 \tag{3.11}$$

Since the goal is to maximize the user gaming experience, the finite time horizon assumption may be not valid anymore. An infinite loop is now possible. For this reason, the discount factor γ has to be set to a value $\gamma < 1$. This value is set to weight the AI player's preference between short-term and long-term rewards.

3.5 Conclusion

In this paper, we presented a modified version of the classic Snakes and Ladders board game. This game is used in our introductory course in artificial intelligence to teach Markov Decision Theory to undergraduate computer science students. Although the game is simple, it contains many interesting aspects also present in more complex computer games. This game offers several perspectives which require different MDP formulations and strategies to generate optimal policies. The most complex challenge is the generation of a policy which optimizes the gaming experience of a human player.

As future work, we consider a few possible avenues to add new challenges for the development of AI algorithms in this game framework. One of them is to use machine learning techniques to automatically learn the opponent's gaming experience model.

For instance, we could provide a database of previously-played games where each game is associated with a score (win, loss, draw, abandonment, etc.) A model could be learned from this history and then be integrated into the MDP policy generation algorithm.

Chapitre 4

Planification des déplacements d'un robot omnidirectionnel et non holonome

Résumé

Les robots omnidirectionnels sont des robots qui peuvent se déplacer naturellement dans toutes les directions, et ce, sans avoir à modifier leur orientation. La modélisation de leur dynamique et leur contrôle représentent plusieurs défis. Les mouvements de ces robots peuvent être modélisés relativement à un centre instantané de rotation (ICR). Ainsi, le contrôle du robot peut être réalisé en spécifiant un ICR et une vitesse de déplacement autour de ce dernier. L'article présente un planificateur de mouvements pour générer des trajectoires efficaces pour ce type de robot dans des environnements contenant des obstacles statiques. L'espace d'états est modélisé par un ICR et une posture. L'algorithme de planification est basé sur l'exploration rapide d'arbres aléatoires, ou *Rapidly-Exploring Random Trees* (RRT), qui échantillonne l'espace des actions pour trouver une trajectoire reliant la configuration initiale à la configuration désirée. Pour générer des trajectoires efficaces, la méthode d'échantillonnage prend en considération des contraintes sur l'ICR afin de réduire les arrêts du robot.

Commentaires

Cet article a été présenté à la *IEEE/RSJ International Conference on Intelligent Robots and Systems* (IROS-2010) [22]. Les travaux ont été initiés à la session d'hiver 2009, où Éric Beaudry et Froduald Kabanza ont encadré Simon Chamberland et Daniel Castonguay dans le cadre d'un cours projet au Département d'informatique. Le projet avait pour but d'identifier les approches et les algorithmes appropriés pour la planification de trajectoires pour AZIMUT-3, un robot omnidirectionnel conçu au laboratoire IntRoLab de la Faculté de génie de l'Université de Sherbrooke. Le projet a également mis à contribution Lionel Clavien et Michel Lauria du laboratoire IntRoLab. Suite à ce projet, une solution de planification de mouvements a été développée pour le robot AZIMUT-3.

L'auteur de la présente thèse, Éric Beaudry, est le deuxième auteur de l'article. Il a participé à la direction des travaux, aux idées ainsi qu'à la rédaction de l'article. Le projet a été supervisé par les professeurs Froduald Kabanza, François Michaud et Michel Lauria. La contribution de cet article est complémentaire aux trois premiers chapitres. Les approches de planification présentées dans les premiers chapitres sont essentiellement des planificateurs de tâches. Ils sont également basés sur des recherches dans des espaces d'états.

Généralement, dans une architecture robotique, la planification de tâches et de trajectoires se font à l'aide de deux planificateurs distincts. Cette séparation est souvent requise pour simplifier le problème de planification. Les planificateurs de tâches et de trajectoires collaborent : le premier décide de l'endroit où aller, alors que le second décide du chemin ou de la trajectoire à emprunter. Comme indiqué dans l'introduction, pour planifier, un planificateur de tâches a besoin de simuler les conséquences de ses actions. Par exemple, le premier planificateur a besoin d'estimer la durée des déplacements. Une façon d'estimer la durée des déplacements est de planifier une trajectoire pour ces derniers.

Droits de reproduction

L'article présenté dans ce chapitre est une copie quasi intégrale de la version publiée. Suite aux recommandations du jury, quelques modifications mineures ont été apportées. Il est à noter que les droits de reproduction du matériel présenté dans ce chapitre sont détenus par IEEE. En date du 5 mars 2011, une permission de reproduction a été accordée à l'auteur par IEEE afin d'intégrer l'article dans cette thèse. Cette permission permet à l'auteur, à l'Université de Sherbrooke et à Bibliothèque et Archives Canada de distribuer des copies du présent chapitre.

Motion Planning for an Omnidirectional Robot With Steering Constraints [1]

Simon Chamberland, Éric Beaudry, Froduald Kabanza
Département d'informatique, Université de Sherbrooke,
Sherbrooke, Québec, Canada J1K 2R1
simon.chamberland@usherbrooke.ca,
eric.beaudry@usherbrooke.ca, froduald.kabanza@usherbrooke.ca
Lionel Clavien, François Michaud
Département de génie électrique et de génie informatique,
Université de Sherbrooke,
Sherbrooke, Québec, Canada J1K 2R1
lionel.clavien@usherbrooke.ca,
francois.michaud@usherbrooke.ca
Michel Lauria
University of Applied Sciences Western Switzerland (HES-SO),
Genève, Suisse
michel.lauria@hesge.ch

Abstract

Omnidirectional mobile robots, i.e., robots that can move in any direction without changing their orientation, offer better manoeuvrability in natural environments. Modeling the kinematics of such robots is a challenging problem and different approaches have been investigated. One of the best approaches for a nonholonomic robot is to model the robot's velocity state as the motion around its instantaneous center of rotation (ICR). In this paper, we present a motion planner designed to compute efficient trajectories for such a robot in an environment with obstacles. The action space is modeled in terms of changes of the ICR and the motion around it. Our motion planner is based on a Rapidly-Exploring Random Trees (RRT) algorithm to

sample the action space and find a feasible trajectory from an initial configuration to a goal configuration. To generate fluid paths, we introduce an adaptive sampling technique taking into account constraints related to the ICR-based action space.

4.1 Introduction

Many real and potential applications of robots include exploration and operations in narrow environments. For such applications, omnidirectional mobile platforms provide easier manoeuvrability when compared to differential-drive or skid-steering platforms. Omnidirectional robots can move sideways or drive on a straight path without changing their orientation. Translational movement along any desired path can be combined with a rotation, so that the robot arrives to its destination with the desired heading.

Our interest lies in nonholonomic omnidirectional wheeled platforms, which are more complex to control than holonomic ones. This complexity stems from the fact that they cannot instantaneously modify their velocity state. Nevertheless, nonholonomic robots offer several advantages motivating their existence. For instance, the use of conventional steering wheels reduces their cost and results in a more reliable odometry, which is important for many applications. Our work is centered around AZIMUT-3, the third prototype of AZIMUT [50, 37], a multi-modal nonholonomic omnidirectional platform. The wheeled configuration of AZIMUT-3, depicted on Figure 4.1, is equipped with four wheels constrained to steer over a 180° range, and a passive suspension mechanism.

There are different approaches to control the kinematics of a wheeled omnidirectional robot. For AZIMUT, we chose to model the velocity state by the motion around the robot's instantaneous center of rotation (ICR) [20]. The ICR is defined as the unique point in the robot's frame which is instantaneously not moving with respect to the robot. For a robot using conventional steering wheels, this corresponds to the point where the propulsion axis of each wheel intersect.

The robot's chassis represents a physical constraint on the rotation of the wheels around their steering axis. These constraints introduce discontinuities on the steering angle of some wheels when the ICR moves continuously around the robot. In fact, a small change of the ICR position may require reorienting the wheels, such that at least one wheel has to make a full 180° rotation. This rotation takes some time, depending on the steering axis' maximum rotational speed. During such wheel reorientation, the ICR is undefined, and because the robot is controlled through its ICR, it must

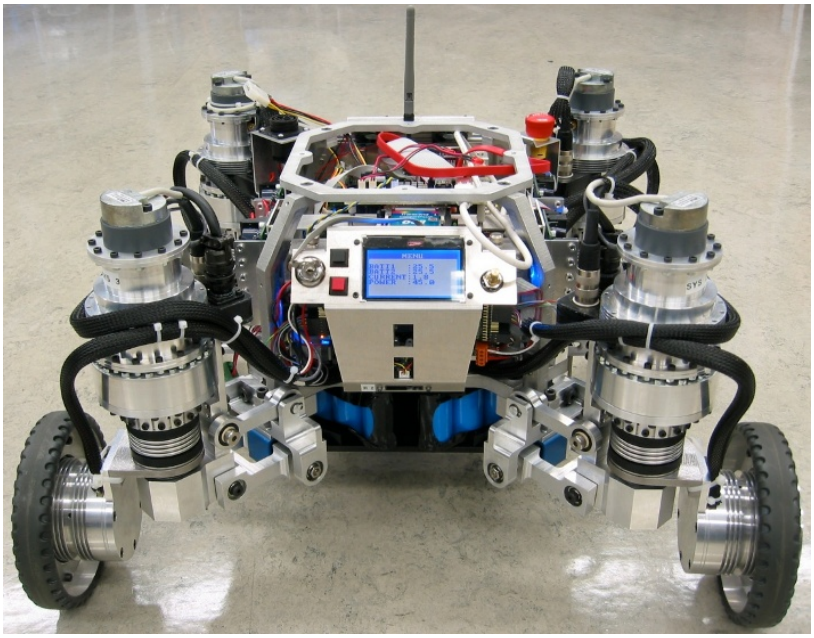


Figure 4.1: The AZIMUT-3 platform in its wheeled configuration

be stopped until the wheel reorientation is completed and the new ICR is reset.

As a solution, a motion planner for an ICR-based motion controller could return a trajectory avoiding wheel reorientations as much as possible, in order to optimize travel time and keep fluidity in the robot's movements. One way to achieve this is to use a traditional obstacle avoidance path planner ignoring the ICR-related kinematic constraints, and then heuristically smoothing the generated path to take into account the constraints that were abstracted away. This is in fact one of the approaches used to reduce intrinsically nonholonomic motion planning problems to holonomic ones [39].

However, we believe this approach is not well suited for AZIMUT, as we would prefer to perform a global optimization of the trajectories, instead of optimizing a potentially ill-formed path. To this end, we chose to investigate another approach which takes directly into account the kinematic constraints related to the ICR and to the robot's velocity. The action space of the robot is modeled as the space of possible ICR changes. We adopt a Rapidly-Exploring Random Trees (RRT) planning approach [38, 35, 39] to sample the action space and find a feasible trajectory from

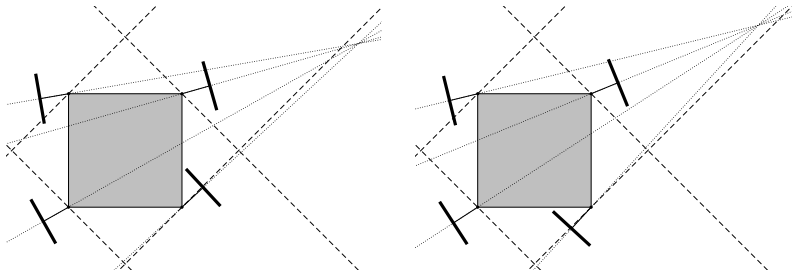
an initial configuration to a goal configuration. To generate fluid paths, we introduce an adaptive sampling technique taking into account the constraints related to the ICR-based action space.

The rest of the paper is organized as follows. Sect. 4.2 describes the velocity state of AZIMUT-3 and Sect. 4.3 characterizes its state space. Sect. 4.4 presents a RRT-based motion planner which explicitly considers the steering limitations of the robot, and Sect. 4.5 concludes the paper with simulation results.

4.2 Velocity State of AZIMUT

Two different approaches are often used to describe the velocity state of a robot chassis [20]. The first one is to use its twist (linear and angular velocities) and is well adapted to holonomic robots, because their velocity state can change instantly (ignoring the maximum acceleration constraint). However, this representation is not ideal when dealing with nonholonomic robots, because their instantaneously accessible velocities from a given state are limited (due to the non-negligible reorientation time of the steering wheels). In these cases, modeling the velocity state using the rotation around the current ICR is preferred.

As a 2D point in the robot frame, the ICR position can be represented using two independent parameters. One can use either Cartesian or polar coordinates to do so, but singularities arise when the robot moves in a straight line manner (the ICR thus lies at infinity). An alternative is to represent the ICR by its projection on a unit sphere tangent to the robot frame at the center of the chassis [23]. This can be visualized by tracing a line between the ICR in the robot frame and the sphere center. Doing so produces a pair of antipodal points on the sphere's surface, as shown on Figure 4.4. Using this representation, an ICR at infinity is projected onto the sphere's equator. Therefore, going from one near-infinite position to another (e.g., when going from a slight left turn to a slight right turn) simply corresponds to an ICR moving near the equator of the sphere. In the following, we define $\boldsymbol{\lambda} = (u; v; w)$ as the 3D Cartesian position of the ICR on the sphere [23] and $\mu \in [-\mu_{max}; \mu_{max}]$ as the motion around that ICR, with μ_{max} being the fastest allowable motion. The whole velocity state is then defined as $\boldsymbol{\eta} = (\boldsymbol{\lambda}; \mu)$.



(a) ICR at $(0.8; 0.43)$ (b) ICR at $(0.8; 0.51)$

Figure 4.2: ICR transition through a steering limitation. Observe the 180° rotation of the lower right wheel.

AZIMUT has steering limitations for its wheels. These limitations define the injectivity of the relation between ICR ($\boldsymbol{\lambda}$) and wheel configurations ($\boldsymbol{\beta}$, the set of all N steering angles). If there is no limitation, one ICR corresponds to 2^N wheel configurations, where $N \geq 3$ is the number of steering wheels. If the limitation is more than 180°, some ICR are defined by more than one wheel configuration. If the limitation is less than 180°, some ICR cannot be defined by a wheel configuration. It is only when the limitation is of 180°, as it is the case with AZIMUT, that the relation is injective: for each given ICR, there exists a unique wheel configuration.

Those limitations can hinder the motion of the ICR. Indeed, each limitation creates a frontier in the ICR space. When a moving ICR needs to cross such a frontier, one of the wheel needs to be "instantly" rotated by 180°. One example of such a situation for AZIMUT-3 is shown on Figure 4.2. To facilitate understanding, the ICR coordinates are given in polar form. As the ICR needs to be defined to enable motion, the robot has to be stopped for this rotation to occur. As shown on Figure 4.3, the set of all limitations splits up the ICR space into several zones, and transitions between any of these zones are very inefficient. In the following, we refer to these ICR control zones as "modes".

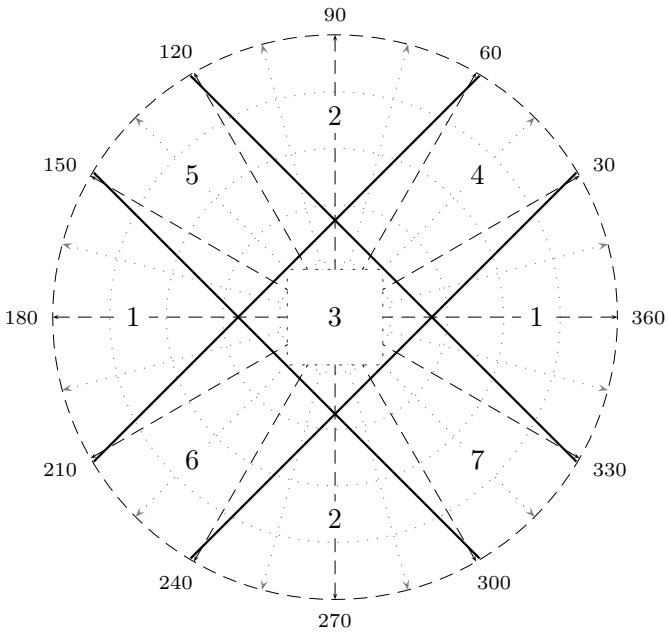


Figure 4.3: Different control zones (modes) induced by the steering constraints. The dashed square represents the robot without its wheels.

4.3 Planning State Space

The representation of states highly depends on the robot to control. A state $\boldsymbol{s} \in S$ of AZIMUT-3 is expressed as $\boldsymbol{s} = (\boldsymbol{\xi}; \boldsymbol{\lambda})$, where $\boldsymbol{\xi} = (x; y; \theta) \in \mathbb{SE}(2)$ represents the posture of the robot and $\boldsymbol{\lambda}$ is its current ICR, as shown on Figure 4.4.

As mentioned in Sect. 4.2, the ICR must be defined at all times during motion for the robot to move safely. Keeping track of the ICR instead of the steering axis angles therefore prevents expressing invalid wheel configurations. Since the relation $\boldsymbol{\lambda} \mapsto \boldsymbol{\beta}$ is injective, any given ICR corresponds to a unique wheel configuration, which the motion planner can disregard by controlling the ICR instead. Because AZIMUT can accelerate from zero to its maximal speed in just a few tenths of a second, the acceleration is not considered by the planner. Thus, instantaneous changes in velocity (with no drift) are assumed, which is why the current velocity of the robot is not included in the state variables.

A trajectory σ is represented as a sequence of n pairs $((\boldsymbol{u}_1, \Delta t_1), \ldots, (\boldsymbol{u}_n, \Delta t_n))$

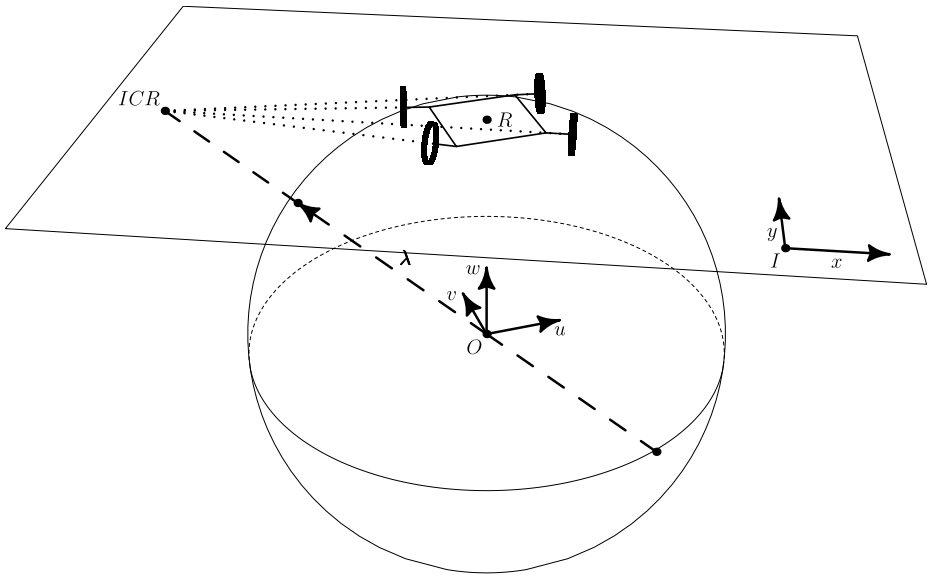


Figure 4.4: State parameters. $R(0;0;1)$ is the center of the chassis.

where $\boldsymbol{u_i} \in U$ is an action vector applied for a duration Δt_i. In our case, the action vector corresponds exactly to the velocity state to transition to: $\boldsymbol{u} = \boldsymbol{\eta}_u = (\boldsymbol{\lambda}_u; \mu_u)$, respectively the new ICR to reach and the desired motion around that ICR.

It is necessary to have a method for computing the new state $\boldsymbol{s'}$ arising from the application of an action vector $\boldsymbol{u}$ for a certain duration Δt to a current state $\boldsymbol{s}$. Since the state transition equation expressing the derivatives of the state variables is non-linear and complex, we use AZIMUT's kinematical simulator as a black box to compute new states. The simulator implements the function $K : S \times U \to S = \boldsymbol{s'} \mapsto K(\boldsymbol{s}, \boldsymbol{u})$ via numerical integration.

4.4 Motion Planning Algorithm

Following the RRT approach [38, 35], our motion planning algorithm (Alg. 11) expands a search tree of feasible trajectories until reaching the goal. The initial state of the robot $\boldsymbol{s}_{init}$ is set as the root of the search tree. At each iteration, a random state $\boldsymbol{s}_{rand}$ is generated (Line 4). Then its nearest neighboring node $\boldsymbol{s}_{near}$ is computed (Line 5), and an action is selected (Line 6) which, once applied from $\boldsymbol{s}_{near}$, produces

an edge extending toward the new sample $\boldsymbol{s}_{rand}$. A local planner (Line 7) then finds the farthest collision-free state $\boldsymbol{s}_{new}$ along the trajectory generated by the application of the selected action. The problem is solved whenever a trajectory enters the goal region, C_{goal}. As the tree keeps growing, so does the probability of finding a solution. This guarantees the probabilistic completeness of the approach.

Algorithm 11 RRT-Based Motion Planning Algorithm

```
1.  RRT-Planner(s_init, s_goal)
2.    T.init(s_init)
3.    repeat until time runs out
4.      s_rand ← GenerateRandomSample()
5.      s_near ← SelectNodeToExpand(s_rand, T)
6.      (u, Δt) ← SelectAction(s_rand, s_near)
7.      s_new ← LocalPlanner(s_near, u, Δt)
8.      add s_new to T.Nodes
9.      add (s_near, s_new, u) to T.Edges
10.     if C_goal is reached
11.       return Extract-trajectory(s_new)
12.   return failure
```

The fundamental principle behind RRT approaches is the same as probabilistic roadmaps [58]. A naïve state sampling function (e.g., uniform sampling of the state space) loses efficiency when the free space C_{free} contains narrow passages – a narrow passage is a small region in C_{free} in which the sampling density becomes very low. Some approaches exploit the geometry of obstacles in the workspace to adapt the sampling function accordingly [62, 36]. Other approaches use machine learning techniques to adapt the sampling strategy dynamically during the construction of the probabilistic roadmap [19, 34].

For the ICR-based control of AZIMUT, we are not just interested in a sampling function guaranteeing probabilistic completeness. We want a sampling function that additionally improves the travel time and motion fluidity. The fluidity of the trajectories is improved by minimizing:

- the number of mode switches;
- the number of reverse motions, i.e., when the robot goes forward, stops, then backs off. Although these reverse motions do not incur mode switches, they are obviously not desirable.

4.4.1 Goal and Metric

The objective of the motion planner is to generate fluid and time-efficient trajectories allowing the robot to reach a goal location in the environment. Given a trajectory $\sigma = ((\boldsymbol{u}_1, \Delta t_1), \ldots, (\boldsymbol{u}_n, \Delta t_n))$, we define reverse motions as consecutive action pairs $(\boldsymbol{u}_i, \boldsymbol{u}_{i+1})$ where $|\tau_i - \tau_{i+1}| \geq \frac{3\pi}{4}$, in which $\tau_i = \arctan(\boldsymbol{\lambda}_{v,i}, \boldsymbol{\lambda}_{u,i}) - \text{sign}(\mu_i)\frac{\pi}{2}$ represents the approximate heading of the robot. Let $(\boldsymbol{s}_0, \ldots, \boldsymbol{s}_n)$ denote the sequence of states produced by applying the actions in σ from an initial state $\boldsymbol{s}_0$. Similarly, we define mode switches as consecutive state pairs $(\boldsymbol{s}_i, \boldsymbol{s}_{i+1})$ where $\text{mode}(\lambda_i) \neq \text{mode}(\lambda_{i+1})$. To evaluate the quality of trajectories, we specify a metric $q(\sigma)$ to be minimized as

$$q(\sigma) = t + c_1 m + c_2 r \tag{4.1}$$

where $c_1, c_2 \in \mathbb{R}^+$ are weighting factors; t, m and r are, respectively, the duration, the number of mode switches and the number of reverse motions within the trajectory σ.

4.4.2 Selecting a Node to Expand

Line 4 of Alg. 11 generates a sample $\boldsymbol{s}_{rand}$ at random from a uniform distribution. As it usually allows the algorithm to find solutions faster [39], there is a small probability P_g of choosing the goal state instead.

Line 5 selects an existing node $\boldsymbol{s}_{near}$ in the tree to be extended toward the new sample $\boldsymbol{s}_{rand}$. Following a technique introduced in [32], we alternate between two different heuristics to choose this node.

Before a feasible solution is found, the tree is expanded primarily using an *exploration* heuristic. This heuristic selects for expansion the nearest neighbor of the sample $\boldsymbol{s}_{rand}$, as the trajectory between them will likely be short and therefore require few collision checks. Hence $\boldsymbol{s}_{near} = \underset{\boldsymbol{s}_i \in T.Nodes}{\arg\min}\ D_{exp}(\boldsymbol{s}_i, \boldsymbol{s}_{rand})$.

The distance metric used to compute the distance between two states $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$

is specified as

$$\begin{aligned} D_{exp}(\boldsymbol{s}_1, \boldsymbol{s}_2) = &\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \\ &+ \frac{1}{\pi} |\theta_2 - \theta_1| \\ &+ \frac{1}{2\pi} \sum_{j \in [1;4]} |\beta_{2,j} - \beta_{1,j}| \end{aligned} \tag{4.2}$$

which is the standard 2D Euclidean distance extended by the weighted sum of the orientation difference and the steering angles difference.

This heuristic is often used in the RRT approach as it allows the tree to rapidly grow toward the unexplored portions of the state space. In fact, the probability that a given node be expanded (chosen as the nearest neighbor) is proportional to the volume of its Voronoi region [63]. Nodes with few distant neighbors are therefore more likely to be expanded.

Once a solution has been found, more emphasis is given to an *optimization* heuristic which attempts to smooth out the generated trajectories. We no longer select the sample's nearest neighbor according to the distance metric D_{exp} (4.2). Instead, nodes are sorted by the weighted sum of their cumulative cost and their estimated cost to $\boldsymbol{s}_{rand}$. Given two samples $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$, we define this new distance as:

$$D_{opt}(\boldsymbol{s}_1, \boldsymbol{s}_2) = q(\sigma_{s_1}) + c_3\, h^2(\boldsymbol{s}_1, \boldsymbol{s}_2) \tag{4.3}$$

where $q(\sigma_{s_1})$ is the cumulative cost of the trajectory from the root configuration to $\boldsymbol{s}_1$ (see (4.1)), $h(\boldsymbol{s}_1, \boldsymbol{s}_2)$ is the estimated cost-to-go from $\boldsymbol{s}_1$ to $\boldsymbol{s}_2$, and $c_3 \in \mathbb{R}^+$ is a weighting factor. We select $\boldsymbol{s}_{near}$ as the node with the lowest distance D_{opt} to $\boldsymbol{s}_{rand}$, or more formally $\boldsymbol{s}_{near} = \underset{\boldsymbol{s}_i \in T.Nodes}{\arg\min}\ D_{opt}(\boldsymbol{s}_i, \boldsymbol{s}_{rand})$.

We set $h(\boldsymbol{s}_1, \boldsymbol{s}_2)$ as a lower bound on the travel duration $\boldsymbol{s}_1$ to $\boldsymbol{s}_2$, which is found by computing the time needed to reach $\boldsymbol{s}_2$ via a straight line at maximum speed, i.e.

$$h(\boldsymbol{s}_1, \boldsymbol{s}_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} / \mu_{max} \tag{4.4}$$

Since $h(\boldsymbol{s}_1, \boldsymbol{s}_2)$ is a lower bound, the cumulative cost $q(\sigma_{s_1})$ and the cost-to-go $h(\boldsymbol{s}_1, \boldsymbol{s}_2)$ cannot contribute evenly to the distance $D_{opt}(\boldsymbol{s}_1, \boldsymbol{s}_2)$. If this was the case, the closest

node (in the D_{opt} sense) to an arbitrary node $\boldsymbol{s}_j$ would always be the root node, as

$$h(\boldsymbol{s}_{root}, \boldsymbol{s}_j) \leq q(\sigma_{s_i}) + h(\boldsymbol{s}_i, \boldsymbol{s}_j) \quad \forall \boldsymbol{s}_i, \boldsymbol{s}_j \tag{4.5}$$

Instead, we give $h(\boldsymbol{s}_1, \boldsymbol{s}_2)$ a quadratic contribution to the total distance. This is meant to favor the selection of relatively close nodes, as the total cost rapidly increases with the distance.

When the objective is to find a feasible solution as quickly as possible, the optimization heuristic (4.3) is not used and the algorithm rather relies on the standard exploration heuristic (4.2). On the other hand, when the algorithm is allocated a fixed time window, it uses both heuristics at the same time. Indeed, prior to finding a solution, the exploration heuristic has a higher likelihood of being chosen, while the optimization heuristic is selected more often once a solution has been found. Combining both heuristics is useful, as performing optimization to improve the quality of the trajectories can be beneficial even before a solution is found. Similarly, using the exploration heuristic once a solution has been computed can sometimes help in finding a shortest path which was initially missed by the algorithm.

4.4.3 Selecting an Action

Line 6 of Alg. 11 selects an action $\boldsymbol{u}$ with a duration Δt which, once applied from the node $\boldsymbol{s}_{near}$, hopefully extends the tree toward the target configuration $\boldsymbol{s}_{rand}$. Obstacles are not considered here.

Since we know more or less precisely the trajectory followed by the robot when a certain action $\boldsymbol{u}$ is given, we can sample the action space with a strong bias toward efficient action vectors. Note that the robot's velocity μ_u should be reduced in the immediate vicinity of obstacles. For now, we disregard this constraint as we always set $\mu_u = \pm\mu_{max}$, which instructs the robot to constantly travel at its maximum velocity. Additionally, we do not require the robot's orientation θ to be tangent to the trajectory.

Let $\boldsymbol{p}_{near} = (x_{near}; y_{near})$ and $\boldsymbol{p}_{rand} = (x_{rand}; y_{rand})$ be the coordinates of the chassis position of respectively $\boldsymbol{s}_{near}$ and $\boldsymbol{s}_{rand}$. Given $\boldsymbol{p}_{near}$ and $\boldsymbol{p}_{rand}$, we can draw an infinite number of circles passing through the two points. Each circle expresses two different

curved trajectories (two exclusive arcs) which can be followed by the robot to connect to the target configuration, assuming the robot's wheels are already aligned toward the circle's center. In this context, the sign of μ_u determines which arc (the smallest or the longest) the robot will travel on. All the centers of these circles lie on the bisector of the $[\boldsymbol{p}_{near};\boldsymbol{p}_{rand}]$ segment, which can be expressed in parametric form as

$$l_\lambda(k) = \frac{1}{2}(\boldsymbol{p}_{near} - \boldsymbol{p}_{rand}) + k\boldsymbol{v} \tag{4.6}$$

where $\boldsymbol{v} \cdot (\boldsymbol{p}_{near} - \boldsymbol{p}_{rand}) = 0$. This line therefore represents the set of ICR allowing a direct connection to the target configuration.

However, some of these ICR should be preferred over others, to avoid mode switches whenever possible. For this purpose, we restrain l_λ to the segment enclosing all ICR in the same mode as the source ICR, i.e., we find all k where $\text{mode}(l_\lambda(k)) = \text{mode}(\boldsymbol{\lambda}_{near})$. Doing so involves computing the intersection between l_λ and the four lines delimiting the different modes (see Figure 4.3). If such a segment exists, we can sample directly a value $k_u \in]k_{min}; k_{max}[$ and set $\boldsymbol{\lambda}_u = l_\lambda(k_u)$, an acceptable ICR which avoids switching to another mode. However, this approach is not adequate, as all 2D points along the segment have equal likelihood of being selected. Indeed, we would prefer faraway points to have less coverage, since each of them corresponds to negligible variations of the wheel angles, and therefore negligible variations of the trajectories.

We address this problem by sampling an ICR on the unit sphere instead (depicted on Figure 4.4). This is achieved by projecting the line segment on the sphere, which yields an arc of a great circle that can be parameterized by an angle $\lambda = \lambda(\phi)$, where $\phi \in [\phi_{min}; \phi_{max}]$. We then sample uniformly $\phi_u = Un(\phi_{min}, \phi_{max})$, from which we compute directly the desired ICR $\lambda_u = \lambda(\phi_u)$. By considering the relation between this angle and its corresponding point back on the global plane, one can see that the farther the point lies from the robot, the less likely it is to be selected. Sampling an angle along a great circle instead of a point on a line segment therefore provides a more convenient ICR probability distribution.

Since we determined the values of λ_u and $|\mu_u|$, what remains to be decided are the sign of μ_u and Δt, the duration of the trajectory. The sign of μ_u is simply set

as to always generate motions along the smallest arc of circle. We then calculate the duration of the path as the time needed to connect from $\boldsymbol{p}_{near}$ to $\boldsymbol{p}_{rand}$ along this arc of circle centered at $\boldsymbol{\lambda}_u$, given $|\mu_u| = \mu_{max}$.

An overview of the algorithm is presented in Alg. 12. Note that besides introducing an additional probability P_l of generating straight lines, we allowed "naïve" sampling to take place with a finite probability P_n, i.e., sampling an ICR without any consideration for the modes.

Algorithm 12 SelectAction Algorithm

```
1.  SelectAction(s_rand, s_near)
2.    if Un(0,1) < P_l
3.      λ_u ← (u; v; 0) the ICR lying at infinity
4.    else
5.      find l_λ(k) = k v + m
6.      project l_λ(k) on the sphere, yielding λ(θ) = (u, v, w)
7.      if Un(0,1) < P_n
8.        λ_u ← λ(Un(0, π))
9.      else
10.       find [θ_min, θ_max] such that
                ∀_{θ∈[θ_min,θ_max]} mode(λ(θ)) = mode(λ_{s_near})
11.       if ∄θ | mode(λ(θ)) = mode(λ_{s_near})
12.         λ_u ← λ(Un(0, π))
13.       else
14.         λ_u ← λ(Un(θ_min, θ_max))
15.   μ_u ← ±μ_max depending on λ_u
16.   Δt ← time to reach s_rand when following the circle arc
17.   return (λ_u; μ_u) and Δt
```

4.5 Results

Pending implementation on AZIMUT, experiments were performed within the OOPSMP [2] [55] library. Since the actual robot and our simulator both use the same kinematical model to compute the robot's motion, we expect the results obtained in simulation to be somewhat similar to real-world scenarios.

Table 4.1 summarizes the values used for the parameters described in (4.1), (4.3) and the different probabilities. The weighting factors c_1 and c_2 were both set to

2. http://www.kavrakilab.rice.edu

Table 4.1: Parameters used

$q(\sigma)$ (4.1)		D_{opt} (4.3)			
c_1	c_2	c_3	P_g	P_l	P_n
2.5	2.5	0.5	0.025	0.25	0.1

(a) Env #1 (5 seconds) (b) Env #2 (10 seconds) (c) Env #3 (25 seconds) (d) Env #4 (20 seconds)

Figure 4.5: Environments and time allocated for each query

2.5, which means every mode switch or reverse motion contributes an additional 2.5 seconds to the total cost. The exploration heuristic is selected 70% of the time prior to finding a solution, and 20% of the time after a solution has been found.

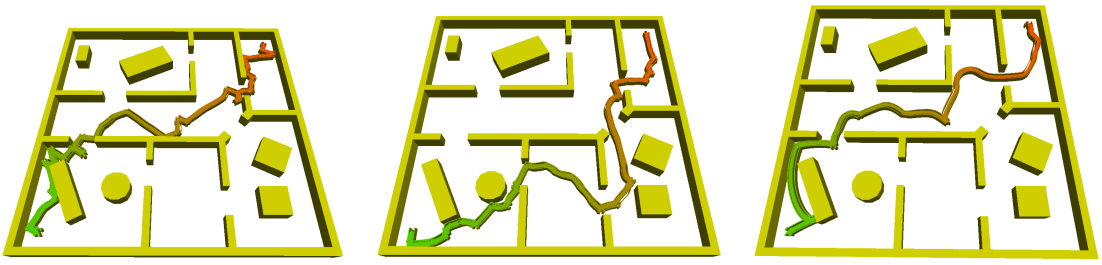
We compared our approach with a "naïve" algorithm ignoring mode switches and reverse motions. This naïve algorithm is a degenerate case of our main one, in the sense that it minimizes the metric (4.1) under the special condition $c_1 = c_2 = 0$ (duration only), and always selects an action naïvely, i.e., with $P_n = 1$. Other parameters remain the same.

An Intel Core 2 Duo 2.6 GHz with 4 GB of RAM was used for the experiments. The results are presented in Table 4.2. Exactly 50 randomly generated queries had to be solved within each environment, with a fixed time frame allocated for each query. However, the time allocated was not the same for each environment, as some are more complicated than others (see Figure 4.5) and we wanted to maximize the number of queries successfully solved.

The results show that the biased algorithm outperformed the naïve one on all environments. Indeed, by minimizing the number of mode switches and reverse motions, the biased algorithm not only improves the fluidity of the trajectories, but also

Table 4.2: Comparison of a naïve algorithm and our proposed solution

	Algorithm	Travel Time	Mode switches	Reverse motions	Metric evaluation (4.1)
Env #1	Naïve	36.36	4.71	0.71	49.91
	Biased	32.41	2.65	0.46	40.19
Env #2	Naïve	38.09	6.46	0.46	55.39
	Biased	33.46	3.41	0.45	43.11
Env #3	Naïve	48.70	12.96	1.19	84.08
	Biased	45.18	10.48	1.92	76.18
Env #4	Naïve	60.42	5.18	0.84	75.47
	Biased	51.72	3.29	0.52	61.25



(a) Trajectory using a completely random sampling

(b) Trajectory using a naïve sampling

(c) Trajectory using our proposed sampling

Figure 4.6: Comparison of trajectories created by a random, a naïve, and a biased algorithms

decreases the average travel time. In heavily cluttered environments like Env #3, feasible trajectories are impaired by a large number of mode switches. To avoid these mode switches, the biased algorithm had to increase the number of reverse motions, which explains the slightly worse result for this particular element. Figure 4.6 presents examples of typical trajectories generated by the different algorithms, including an algorithm selecting an ICR randomly. However, it is important to note that our algorithm does not always produce good-looking trajectories, as guarantees of quality are hard to obtain via nondeterministic approaches.

4.6 Conclusion

A new RRT-based algorithm for the motion planning of nonholonomic omnidirectional robots has been presented. It has been shown that by taking explicitly into account the kinematic constraints of such robots, a motion planner could greatly improve the fluidity and efficiency of trajectories.

We plan to expand our RRT-based motion planner to constrain the orientation of the robot's chassis, and to adapt the robot's velocity according to the proximity with obstacles. We are also interested in computing a robust feedback plan so that the robot does not deviate too much from the planned trajectory, despite the inevitable real world unpredictability. For AZIMUT, this would involve the additional challenge of making sure the ICR stays as far as possible from the control zones frontiers, as to avoid undesired mode switches. Future work will integrate these additional elements.

Acknowledgments

This work is funded by the Natural Sciences and Engineering Research Council of Canada, the Canada Foundation for Innovation and the Canada Research Chairs. F. Michaud holds the Canada Research Chair in Mobile Robotics and Autonomous Intelligent Systems.

Conclusion

Les travaux présentés dans cette thèse s'inscrivent dans le domaine de la planification en intelligence artificielle (IA). L'objectif de ces travaux était d'améliorer la prise de décisions pour une classe de problèmes de planification particulière, soit celle qui combine des actions concurrentes (simultanées) et de l'incertitude. Bien que ces deux aspects ont été largement étudiés séparément, leur combinaison représente des défis considérables.

La classe de problèmes ciblée est motivée par de nombreuses applications réelles. La robotique mobile est un exemple de systèmes intelligents nécessitant une telle capacité. En effet, les robots mobiles évoluent généralement dans des environnements dynamiques et incertains. Lorsqu'ils doivent interagir avec des humains, les actions des robots peuvent avoir des durées incertaines. De plus, ils peuvent souvent exécuter des actions simultanées afin d'être plus efficaces. Cette classe de problèmes a également été identifiée par la NASA comme étant très importante pour la planification des actions des robots déployés sur Mars.

Plusieurs contributions significatives ont été présentées dans cette thèse. Au moment de sa rédaction, une portion importante de celles-ci ont notamment fait l'objet de publications à des conférences scientifiques, dont l'une à l'*International Conference on Automated Planning and Scheduling* (ICAPS) qui est considérée comme la conférence la plus spécialisée dans le domaine.

Le premier chapitre a présenté ActuPlan qui est basé sur une nouvelle approche de planification. Cette contribution apporte des solutions à des problèmes de planification avec de l'incertitude lié au temps (durée des actions) et aux ressources (consommation et production de ressources). La solution proposée utilise une représentation compacte d'états basée sur un modèle continu de l'incertitude. En effet, des variables

aléatoires continues, organisées dans un réseau bayésien construit dynamiquement, sont utilisées pour modéliser le temps et l'état des ressources. ActuPlan^{nc}, le premier planificateur présenté, permet de générer des plans non conditionnels qui sont quasi optimaux optimaux. Ce planificateur a été modifié pour générer un ensemble de plans non conditionnels. Ces plans non conditionnels sont ensuite fusionnés en un plan conditionnel qui retarde certaines décisions au moment de l'exécution. Les conditions de branchement sont établies en conditionnant l'espérance de variables aléatoires.

En plus d'être efficace, la méthode proposée procure plusieurs avantages. Par exemple, contrairement aux approches entièrement basées sur des processus décisionnels markovien (MDP) qui nécessitent une hypothèse markovienne, soit l'indépendance de chaque décision, l'approche proposée offre une plus grande flexibilité sur la modélisation de la dépendance (ou l'indépendance) entre les diverses sources d'incertitude. En effet, puisqu'elle exploite un réseau bayésien, il est possible d'ajouter des dépendances entre les variables aléatoires afin d'être plus fidèle à la réalité.

La deuxième contribution présentée est une généralisation d'ActuPlan à plusieurs formes d'incertitude, incluant celle sur les effets des actions. Celle-ci a mené à la conception du planificateur QuanPlan, un planificateur hybride qui est basé sur deux fondements bien établis en IA. Tandis que l'incertitude sur le temps est prise en charge par un réseau bayésien, l'incertitude sur les effets des actions est prise en charge par un MDP. Afin de modéliser des effets indéterminés, une notion d'état quantique a été introduite. Cette représentation permet de modéliser des superpositions d'états dont leur détermination est retardée au moment de leur observation.

Les nouvelles techniques de planification présentées dans cette thèse ont été validées sur deux domaines d'applications, soit les domaines de transport et des Mars *rovers* qui sont utilisés lors des *International Planning Competitions* qui ont lieu aux deux ou trois ans lors des conférences ICAPS. Ces derniers ont été modifiés afin d'introduire de l'incertitude au niveau de la durée des actions et ainsi que sur les effets. Bien que ces domaines soient artificiels, ils renferment des caractéristiques fondamentales d'applications réelles.

Cette thèse a également exploré un contexte d'application différent, soit celui des jeux. Ce volet est une contribution à l'application de techniques d'IA dans les jeux. Les jeux représentent une autre application où la prise de décisions automatique est

importante. De plus, l'incertitude et des actions simultanées sont souvent des aspects importants dans les jeux. Les solutions qui ont été présentées permettent à un joueur artificiel de prendre des actions selon plusieurs critères. Par exemple, les décisions peuvent être orientées pour compléter le plus rapidement possible le jeu ou pour vaincre un adversaire. Des décisions plus complexes ont également été abordées afin de s'adapter automatiquement au niveau de l'adversaire.

Une autre contribution a été présentée dans le domaine de la planification de trajectoires pour des robots omnidirectionnels. Celle-ci est complémentaire aux précédentes qui portent sur la planification d'actions. Le problème de prise de décisions étant généralement très complexe, celui-ci est généralement décomposé. Ainsi, au lieu de réaliser un seul planificateur pour la planification des actions et des trajectoires, deux planificateur distincts sont généralement mis à contribution.

Les travaux présentés dans cette thèse ouvrent la porte à de nouvelles possibilités pour diverses applications qui ont besoin de capacités d'autonomie et de prise de décisions. En plus de la robotique mobile et des jeux, d'autres applications pourraient en bénéficier. Par exemple, certains systèmes d'aide à la décision utilisent la planification pour recommander des actions à prendre. La concurrence d'actions et l'incertitude sont des aspects omniprésents dans bon nombre d'applications. Jusqu'à présent, beaucoup de systèmes intelligents font abstraction de ces aspects puisque leur considération est trop complexe. Ainsi, les avancées présentées dans cette thèse pourraient repousser les limites de ces applications en améliorant la prise de décisions sous incertitude.

Comme travaux futurs, les algorithmes de planification présentés peuvent être intégrés dans une application réelle. En effet, il est prévu de les intégrer dans le robot Johny-0 en vue d'une éventuelle participation à la compétition *RobotCup@Home*. Bien qu'ils sont à déterminer, les scénarios envisagés renferment de l'incertitude et contiennent potentiellement des actions concurrentes. L'intégration de QuanPlan dans un robot comporte plusieurs défis, comme la spécification automatique des distributions de probabilités. En effet, le robot pourrait apprendre automatiquement le modèle probabiliste de la durée des actions. De plus, durant l'exécution, une estimation en continu des durées d'actions pourrait accroitre la fiabilité du module de surveillance des plans.

Bien que cette thèse présente des avancées pour une classe de problèmes de planification très précise, le problème de prise de décisions en intelligence artificielle demeure un important thème de recherche. L'intégration des algorithmes de planification dans des applications réelles, tels des robots mobiles, peut se heurter à de nombres obstacles. En effet, comme indiqué dans l'introduction, les algorithmes de planification exigent un certain nombre d'hypothèses simplificatrices. Les travaux présentés dans cette thèse n'y font pas exception. Les hypothèses simplificatrices étant souvent contraignantes, des ajustements et des stratégies d'intégration sont généralement requis afin que les algorithmes de planification puissent être appliqués à des problématiques réelles [11]. Les hypothèses simplificatrices peuvent également limiter l'applicabilité des planificateurs. Quelques pistes d'amélioration sont présentés.

L'indissociabilité d'une mission (but) est l'une des hypothèses requises des algorithmes présentés dans cette thèse. Une mission étant parfois impossible à réaliser en raison des contraintes sur les ressources et le temps, un planificateur doit sélectionner un sous-ensemble de buts en plus de générer un plan. Plusieurs travaux sur la satisfaction partielle [60] des buts pourraient servir d'inspiration pour améliorer QuanPlan.

La réactivité représente une autre limite de QuanPlan. En effet, ce planificateur est avant tout construit pour résoudre des problèmes de façon hors-ligne (*off-line planning*). Ce planificateur pourrait être amélioré en utilisant des techniques de planification en tout temps (*anytime planning*) [42].

La stabilité des plans est une caractéristique généralement souhaitée. Dans des environnements dynamiques, et où les buts peuvent évoluer avec le temps, le planificateur doit éviter de changer radicalement ses plans lors de la replanification. Il existe plusieurs façons de mesurer la stabilité des plans, comme de compter le nombre d'actions modifiées [30]. Cette métrique pourrait être modifiée pour considérer d'autres facteurs, comme des contraintes sur des engagements pris suite à un premier plan.

Bien que QuanPlan prenne en charge l'incertitude liée aux actions elles-mêmes, l'hypothèse d'observabilité totale demeure requise. Dans les applications réelles, cette hypothèse n'est pas toujours réaliste à cause de l'imperfection des capteurs. Il est possible d'améliorer QuanPlan en s'inspirant de travaux sur les processus décisionnels markoviens avec observation partielle (POMDP) [54].

Bibliographie

[1] M. Ai-Chang, J. Bresina, L. Charest, J. Hsu, A. K. Jonsson, B. Kanefsky, P. Maldague, P. Morris, K. Rajan et J. Yglesias.
« MAPGEN : Mixed-initiative Planning and Scheduling for the Mars Exploration Rover Missions ».
Intelligent Systems, 19(1):8–12, 2004.

[2] F. Bacchus et M. Ady.
« Planning with Resources and Concurrency : A Forward Chaining Approach ».
Dans *Proc. of the International Joint Conference on Artificial Intelligence*, pages 417–424, 2001.

[3] B. W. Ballard.
« The *-Minimax Search Procedure for Trees Containing Chance Nodes ».
Artificial Intelligence, 21(3):327–350, 1983.

[4] A. Bar-Noy et B. Schieber.
« The Canadian Traveller Problem ».
Dans *Proc. of the ACM-SIAM symposium on Discrete algorithms*, pages 261–270. Society for Industrial and Applied Mathematics, 1991.

[5] A.G. Barto, S.J. Bradtke et S.P. Singh.
« Learning to Act Using Real-Time Dynamic Programming ».
Artificial Intelligence, 72(1–2):81–138, 1995.

[6] É. Beaudry, F. Bisson, S. Chamberland et F. Kabanza.
« Using Markov Decision Theory to Provide a Fair Challenge in a Roll-and-Move Board Game ».
Dans *Proc. of the IEEE Computational Intelligence and Games*, 2010.

[7] É. Beaudry, Y. Brosseau, C. Côté, C. Raïevsky, D. Létourneau, F. Kabanza et F. Michaud.
« Reactive Planning in a Motivated Behavioral Architecture ».
Dans *Proc. of the National Conference on Artificial Intelligence (AAAI)*, pages 1242–1249, 2005.

[8] É. Beaudry, F. Kabanza et F. Michaud.
« Planning for a Mobile Robot to Attend a Conference ».
Dans *Proc. of the Advances in Artificial Intelligence (Canadian Conference on AI)*, pages 48–52, 2005.

[9] É. Beaudry, F. Kabanza et F. Michaud.
« Planning for Concurrent Action Executions Under Action Duration Uncertainty Using Dynamically Generated Bayesian Networks ».
Dans *Proc. of the International Conference on Automated Planning and Scheduling*, pages 10–17, 2010.

[10] É. Beaudry, F. Kabanza et F. Michaud.
« Planning with Concurrency under Resources and Time Uncertainty ».
Dans *Proc. of the European Conference on Artificial Intelligence*, pages 217–222, 2010.

[11] É. Beaudry, D. Létourneau, F. Kabanza et F. Michaud.
« Reactive Planning As a Motivational Source in a Behavior-Based Architecture ».
Dans *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2008.

[12] R. E. Bellman.
Dynamic Programming.
Princeton University Press, Princeton, NJ, 1957.

[13] A. Benaskeur, F. Kabanza et É. Beaudry.
« CORALS : A Real-Time Planner for Anti-Air Defence Operations ».
ACM Transactions on Intelligent Systems and Technology, 1(2):1–21, 2010.

[14] A. Benaskeur, F. Kabanza, É. Beaudry et M. Beaudoin.
« A Probabilistic Planner for the Combat Power Management Problem ».

Dans *Proc. of the International Conference on Automated Planning and Scheduling*, pages 12–19, 2008.
Sept, 22-26, 2008, Acropolis Convention Center, Nice, France.

[15] B. Bonet et H. Geffner.
« Planning As Heuristic Search ».
Artificial Intelligence, 129:5–33, 2001.

[16] B. Bonet et H. Geffner.
« Labeled RTDP : Improving the Convergence of Real-Time Dynamic Programming ».
Dans *Proc. of the International Conference on Automated Planning and Scheduling*, pages 12–31, 2003.

[17] J. Bresina, R. Dearden, N. Meuleau, D. Smith et R. Washington.
« Planning Under Continuous Time and Resource Uncertainty : A Challenge For AI ».
Dans *Proc. of the Conference on Uncertainty in AI*, pages 77–84, 2002.

[18] O. Buffet et D. Aberdeen.
« The Factored Policy-Gradient Planner ».
Artificial Intelligence, 173(5–6):722–747, 2009.

[19] B. Burns et O. Brock.
« Sampling-Based Motion Planning Using Predictive Models ».
Dans *Proc. of the IEEE International Conference on Robotics and Automation*, 2005.

[20] G. Campion, G. Bastin et B. d'Andréa-Novel.
« Structural Properties and Classification of Kinematic and Dynamic Models of Wheeled Mobile Robots ».
IEEE Transactions on Robotics and Automation, 12(1):47–62, 1996.

[21] I. Song Gao Chabini.
« Optimal Routing Policy Problems in Stochastic Time-Dependent Networks : I. Framework and Taxonomy ».
Dans *Proc. of the IEEE Conference on Intelligent Transport System*, page 549, 2002.

[22] S. Chamberland, É. Beaudry, L. Clavien, F. Kabanza, F. Michaud et M. Lauria.
« Motion Planning for an Omnidirectional Robot with Steering Constraints ».
Dans *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010.

[23] L. Clavien, M. Lauria et F. Michaud.
« Instantaneous Centre of Rotation Estimation of an Omnidirectional Mobile Robot ».
Dans *Proc. of the IEEE International Conference on Robotics and Automation*, pages 5435–5440, 2010.

[24] W. Cushing, D. S. Weld, S. Kambhampati, Mausam et K. Talamadupula.
« Evaluating Temporal Planning Domains ».
Dans *Proc. of the International Conference on Automated Planning and Scheduling*, pages 105–112, 2007.

[25] A. Darwiche.
Modeling and Reasoning with Bayesian Networks.
Cambridge University Press, April 2009.

[26] R. Dearden, N. Meuleau, S. Ramakrishnan, D. Smith et R. Washington.
« Incremental Contingency Planning ».
Dans *Proc. of the ICAPS Workshop on Planning under Uncertainty*, 2003.

[27] Y. Dimopoulos, A. Gerevini, P. Haslum et A. Saetti.
« The Benchmark Domains of the Deterministic Part of IPC-5 ».
Dans *Working notes of the 16th International Conference on Automated Planning & Scheduling (ICAPS-06) - 5th International Planning Competition*, 2006.

[28] M.B. Do et S. Kambhampati.
« SAPA : A Scalable Multi-Objective Metric Temporal Planner ».
Journal of Artificial Intelligence Research, 20:155–194, 2003.

[29] R. Fikes et N.J. Nilsson.
« STRIPS : A New Approach to the Application of Theorem Proving to Problem Solving ».

Dans *Proc. of the International Joint Conference on Artificial Intelligence*, pages 608–620, 1971.

[30] M. Fox, A. Gerevini, D. Long et I. Serina.
« Plan Stability : Replanning Versus Plan Repair ».
Dans *Proc. of the International Conference on Automated Planning and Scheduling*, 2006.

[31] M. Fox et D. Long.
« PDDL 2.1 : An Extension to PDDL for Expressing Temporal Planning Domains ».
Journal of Artificial Intelligence Research, 20:61–124, 2003.

[32] E. Frazzoli, M. A. Dahleh et E. Feron.
« Real-Time Motion Planning for Agile Autonomous Vehicles ».
AIAA Journal on Guidance, Control on Dynamics, 25(1):116–129, 2002.

[33] J. Hoffmann et B. Nebel.
« The FF Planning System : Fast Plan Generation Through Heuristic Search ».
Journal of Artificial Intelligence Research, 14:253–302, 2001.

[34] D. Hsu, J.-C. Latombe et H. Kurniawati.
« On the Probabilistic Foundations of Probabilistic Roadmap Planning ».
Dans *Proc. of the International Symposium on Robotics Research*, 2005.

[35] J. J. Kuffner et S. M. LaValle.
« RRT-connect : An efficient approach to single-query path planning ».
Dans *Proc. of the IEEE International Conference on Robotics and Automation*, pages 995–1001, 2000.

[36] H. Kurniawati et D. Hsu.
« Workspace Importance Sampling for Probabilistic Roadmap Planning ».
Dans *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2004.

[37] M. Lauria, I. Nadeau, P. Lepage, Y. Morin, P. Giguère, F. Gagnon, D. Létourneau et F. Michaud.
« Design and Control of a Four Steered Wheeled Mobile Robot ».

Dans *Proc. of the 32nd Annual Conference of the IEEE Industrial Electronics*, pages 4020–4025, 7-10 Nov 2006.

[38] S. M. LaValle.
« Rapidly-Exploring Random Trees : A New Tool for Path Planning ».
Rapport Technique TR : 98-1, Computer Science Dept, Iowa State University, 1998.

[39] S. M. LaValle.
Planning Algorithms.
Cambridge University Press, 2006.

[40] A.M. Law et D.W. Kelton.
Simulation Modeling and Analysis.
McGraw-Hill, deuxième édition, 1997.

[41] R. Liboff.
Introductory Quantum Mechanics.
Addison-Wesley, 2003.

[42] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz et S. Thrun.
« Anytime Search in Dynamic Graphs ».
Artificial Intelligence, 172:1613–1643, September 2008.

[43] I. Little, D. Aberdeen et S. Thiébaux.
« Prottle : A Probabilistic Temporal Planner ».
Dans *Proc. of the National Conference on Artificial Intelligence*, 2005.

[44] G. F. Luger.
Artificial Intelligence : Structures and Strategies for Complex Problem Solving.
Addison Wesley, sixième édition, 2009.

[45] Mausam et D. S. Weld.
« Solving Concurrent Markov Decision Processes. ».
Dans Deborah L. McGuinness et George Ferguson, éditeurs, *Proc. of the National Conference on Artificial intelligence (AAAI)*, pages 716–722, 2004.

[46] Mausam et D. S. Weld.
« Probabilistic Temporal Planning with Uncertain Durations ».
Dans *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 2006.

[47] Mausam et D. S. Weld.
« Concurrent Probabilistic Temporal Planning ».
Journal of Artificial Intelligence Research, 31:33–82, 2008.

[48] B. Medler.
« Views From Atop the Fence : Neutrality in Games ».
Dans *Proc. of the ACM SIGGRAPH Symposium on Video Games*, pages 81–88, 2008.

[49] F. Michaud, C. Cote, D. Letourneau, Y. Brosseau, J. Valin, É. Beaudry, C. Raievsky, A. Ponchon, P. Moisan, P. Lepage, Y. Morin, F. Gagnon, P. Giguère, M. Roux, S. Caron, P. Frenette et F. Kabanza..
« Spartacus Attending the 2005 AAAI Conference ».
Autonomous Robots. Special Issue on the AAAI Mobile Robot Competitions and Exhibition, 22(4):369–383, 2007.

[50] F. Michaud, D. Létourneau, M. Arsenault, Y. Bergeron, R. Cadrin, F. Gagnon, M.-A. Legault, M. Millette, J.-F. Paré, M.-C. Tremblay, P. Lepage, Y. Morin, J. Bisson et S. Caron.
« Multi-Modal Locomotion Robotic Platform Using Leg-Track-Wheel Articulations ».
Autonomous Robots, 18(2):137–156, 2005.

[51] D. Nau, M. Ghallab et P. Traverso.
Automated Planning : Theory & Practice.
Morgan Kaufmann Publishers Inc., 2004.

[52] D.S. Nau, T.C. Au, O. Ilghami, U. Kuter, J.W. Murdock, D. Wu et F. Yaman.
« SHOP2 : An HTN Planning System ».
Journal of Artificial Intelligence Research, 20:379–404, 2003.

[53] B. Nebel.
« On the Compilability and Expressive Power of Propositional Planning Formalisms ».
Journal of Artificial Intelligence Research, 12:271–315, 2000.

[54] J. Pineau, G. Gordon et S. Thrun.
« Anytime Point-Based Approximations for Large POMDPs ».
Journal of Artificial Intelligence Research, 27:335–380, 2006.

[55] E. Plaku, K. Bekris et L. E. Kavraki.
« OOPS for Motion Planning : An Online Open-source Programming System ».
Dans *Proc. of the IEEE International Conference on Robotics and Automation*, pages 3711–3716, 2007.

[56] E. Rachelson, P. Fabiani, F. Garcia et G. Quesnel.
« A Simulation-based Approach for Solving Temporal Markov Problems ».
Dans *Proc. of the European Conference on Artificial Intelligence*, 2008.

[57] S. Russell et P. Norvig.
Artificial Intelligence : A Modern Approach.
Prentice Hall, troisième édition, 2010.

[58] G. Sanchez et J.-C. Latombe.
« A Single-Query Bi-Directional Probabilistic Roadmap Planner with Lazy Collision Checking ».
Dans *Proc. of the International Symposium on Robotics Research*, pages 403–417, 2001.

[59] B. Schwab.
AI Game Engine Programming.
Charles River Media, Inc., deuxième édition, 2008.

[60] D. Smith.
« Choosing Objectives in Over-Subscription Planning ».
Dans *Proc. of the International Conference on Automated Planning and Scheduling*, 2004.

[61] R. S. Sutton, D. Mcallester, S. Singh et Y. Mansour.
« Policy Gradient Methods for Reinforcement Learning with Function Approximation ».
Dans *In Advances in Neural Information Processing Systems 12*, pages 1057–1063. MIT Press, 1999.

[62] J.P. van den Berg et M.H. Overmars.
« Using Workspace Information as a Guide to Non-Uniform Sampling in Probabilistic Roadmap Planners ».
International Journal on Robotics Research, pages 1055–1071, 2005.

[63] A. Yershova, L. Jaillet, T. Simeon et S.M. LaValle.
« Dynamic-Domain RRTs : Efficient Exploration by Controlling the Sampling Domain ».
Dans *Proc. of the IEEE International Conference on Robotics and Automation*, 2005.

[64] H. Younes et R. Simmons.
« Policy Generation for Continuous-Time Stochastic Domains with Concurrency ».
Dans *Proc. of the International Conference on Automated Planning and Scheduling*, 2004.