Planning with tokens :

an approach between satisfaction and optimization

Patrick Fabiani and **Yannick Meiller**[1]

Abstract. This paper presents an initial approach in an attempt to integrate powerful propositional planning tools within a general planning process possibly involving both numerical and symbolic uncertainties. Recently developed propositional approaches to planning allow to build and search efficiently a planning graph. Yet, handling uncertainty and numerical optimization criteria within such frameworks remains difficult. A potentially powerful solution is to combine symbolic reasoning tools with decision theoretic representations. A first proposal is to begin with a colored Petri net representation of the propositional planning domain. Sample formalizations of planning problems and subsequent implementations are presented together with the propagation mechanisms with tokens. The plan is output as a Petri net, so that all information about dependency relations among actions is preserved. The preliminary results on deterministic planning problems seem encouraging. Besides, useful information concerning the planning process, uncertainty or risk can then be attached to the colored tokens and thus travel throughout the planning graph. This is meant to be later exploited for optimization.

1 Introduction

The introduction of sensory or modeling uncertainties in the classical planning paradigm leads to practical difficulties both at representation and solution level [13]. Yet, planning under uncertainty is a key issue in research about agents with autonomous decision making abilities. The topics addressed in [6] easily extend to the general case of autonomous agents which have to deal with rich domains while having uncertain or incomplete models of their environment, possibly involving partial observability or partial predictability. Recent work about a problem of dynamic robot motion planning under uncertainty [7] has shown how computational geometry could be combined to game theory in order to reduce the size of the search space. Though dealing with uncertainty is not the chief topic addressed here, this paper presents an initial step in order to generalize this approach and combine symbolic reasoning with decision-theoretic tools. The proposed approach consist in first designing an efficient classical planner borrowing ideas from Graphplan, to be later extended to allow planning under uncertainty. The present paper is organized in three sections. In section § 2 we first relate this work to other approaches and explain our motivations to use tokens with a discussion about optimization issues. In section § 3 we introduce Petri nets and tokens and show how this formalism can be related to our needs. In section § 4 we focus on the mechanisms of planning with tokens in classical domains, eventually providing perspectives on future work on both classical planning and planning with uncertainty.

[1] ONERA-CERT / DCSD, 2 av. Edouard Belin, F-31055 Toulouse cedex, email: fabiani@cert.fr, meiller@cert.fr

2 Related work

2.1 Combining decision theoretic and classical planning

The theory of games and decision [17, 16] provides an attractive framework for decision making under uncertainty, within which the study of automated sequential decision making under uncertainty can be usefully embedded. Different approaches to decision-theoretic planning have been studied and developed in various domains : see [14] for robot motion planning problems and [4] for a discussion about recent issues about decision-theoretic planning and more specifically around the MDP framework.

On the other hand, solving MDPs or stochastic dynamic programming problems of quite reasonable size remains a task of high computational complexity [15]. The curse of dimensionality appears when partial observability is added (POMDPs) : then, the dimension of the search space is equal to the size of the underlying state space. Algorithms have been developed that can efficiently solve decision-theoretic planning problems of reasonable size : in particular, [4] reviews a number of approaches developed to prune as many branches as possible in the search graph.

More specifically, the work in [3] is remarkable as proposing to reuse ideas from Graphplan [1] for reachability analysis in solving MDPs. Yet, it may not always be easy to draw an adapted MDP-like discretization of the state space from the initial problem definition : in [7] for example, the workspaces of the pursuer and the target robots are both systematically discretized into 1500 possible free states each (to be fine enough). Assuming perfect localization in real time for both robots, and focusing on couples of initial positions for which the pursuer can see the target (an average 40 possibilities), the MDP for the tracking problem with uncertain moves but perfect localization information would have a state space of size 60000. Now with localization uncertainties, the corresponding POMDP has a dimension 60000 : there would be a need for a tailored hierarchical decomposition rather than a systematic discretization. The curse of dimensionality is hopefully circumvented in [7] thanks to Computational Geometry tools, which seems a good idea to generalize whenever equivalent powerful tools exist.

This again, leads to the idea that classical planning methods could be somewhat adapted so as to take uncertainty measures into account at planning time, or alternatively, as already proposed by C. Boutilier [3], in order to build an appropriate search graph and allow a subsequent decision process to deal with it properly. For instance, *PGraphplan* in [2] takes probabilistic actions into account and *Sensory Graphplan* [21] handles uncertainty about the initial state. [10] presents another attempt to extend classical planning methods to dynamic and changing environments. Yet, most of the difficulties and limitations raised by S. Kambhampati in [13] still apply.

2.2 Planning with uncertainties, optimization and forward search

The first point we want to make here is that in classical planning, the problem is one of satisfaction of a sequence of transition conditions leading to a goal termination condition. By contrast, dealing with uncertainties in planning, like in decision-theoretic planning, requires optimization capabilities. This actually is one major difficulty for classical planning algorithms to adapt to problems with uncertainty.

For instance, [2] describes an adaptation of Graphplan for doing contingent planning, considering probabilistic actions - Pgraphplan. Pgraphplan builds the graph basically the same way as Graphplan. However, Graphplan considers that both instances p_1 and p_2 of a same proposition appearing in two separate states s_1 and s_2 are equivalent. For that reason, this proposition is introduced only once in a level of the graph - this is the basis of disjunctive planning. By contrast, this assumption is not valid anymore in Pgraphplan's plan-graph. Indeed, the reachability probabilities concerning s_1 and s_2 may be different, so that p_1, for example, may have a higher probability of reachability. Since we are doing contingent planning, it is important to keep this distinction between p_1 and p_2 in order to make the search for a plan easier. Unfortunately, Pgraphplan does not keep it because it builds the graph basically the same way as Graphplan.

As a consequence, in order to extract a plan, the plan graph has to be searched forward instead of backward. This prevents it from taking advantage of most of the speed-ups of classical Graphplan, because these are related to the coupling between forward constraint propagation and backward search. As pointed out in [2], searching the Pgraphplan's plan-graph backward is much more difficult than it is in classical Graphplan.

Similarly in [10], the author try to reuse Graphplan's ideas for a combination of contingent and conformant planning in an uncertain environment. In their approach though, the actual computation of uncertainty levels requires an unreasonable series of backward and forward computation phases.

The same problem occurs about utility functions or rewards : indeed, a given state can be reached from different trajectories with different utility or cost and one would need to keep some track of it in order to do some optimization. Besides, two different states, with two different utility values may share some common partial description. What value must be assigned to the node corresponding to this partial state when no distinction is made between the underlying states, or sets of states.

2.3 States or features ?

Most of the time, the probabilities, costs, utilities, or expected utilities to be optimized can only be computed via a forward search in the state space : the optimization criteria depend upon the states (or sets of states in the best case) and transitions between those states (resp. sets of states) along the performed sequence of actions. These criteria generally cannot be computed from features of the state, like features described by propositions, which is a real problem for disjunctive planners like Graphplan.

Here is a simple optimization example to fix the ideas. It is adapted from the rocket benchmark domain to fit in this discussion. Consider one rocket and four planets : Venus, Mars, the Moon, and the Earth. Two packages - A and B - are on the Earth, along with the rocket. A has to be delivered on Mars and B on Venus, **minimizing** the cost of transportation. Costs are attached to interplanetary journeys and are summarized in figure 1. How could we adapt Grpahplan to find the best transportation plan ? Consider two possible trips of the rocket :

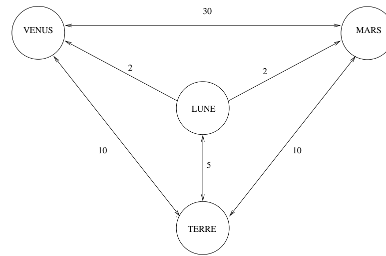


Figure 1. Interplanetary journeys have a cost. Here is an example.

1. Earth - Mars - Moon, $cost = 12$
2. Earth - Moon - (stayed on the Moon), $cost = 5$

While building the plan graph, we observe there are two ways of reaching the moon, with dramatically different cost values (suppose for example that each token wears a cost value that is updated when going through a FLY transition). A temptation is to keep the lowest cost token only. Now suppose while on Mars (in trip 1), the rocket unloaded package A. The higher cost of this trip is then totally justified. In fact, both these costs cannot be compared because they do not correspond to plans of same level of achievement with respect to the final goals to be reached.

To distinguish clearly between the two ways of reaching a node in the plan graph, we need to somewhat keep track of the followed *"trajectory"*, or the sequence of action. The fact is that for this specific domain, propagating tokens in a graph at planning time would allow the tokens to carry all the necessary information for the backward search algorithm to find the best cost afterward, thus naturally solving the problem.

Yet, generalizing this scheme would simply make our approach look very much like a forward search in the state space, and forward search in the state space is extremely costly. We would like to avoid it whenever possible.

2.4 Search space splitting and optimization

As a matter of fact, at each stage of the search space building process, some sets of (reachable) states may share interesting properties. They may have the same utility value - defining *utility regions* among states, like in [7] or they may be reachable with the same probability - defining *probability or risk* regions. In such cases, disjunctive planning, with its capability to manage sets of states (described as propositional features) may have an answer.

This is not always possible : when, on the contrary, the topology of these regions depends on the values assigned to intricate combinations of several features of the domain (thus, defining states), fully disjunctive planner are useless since they consider each feature independently. Intuitively, a nice solution should involve some “splitting” control based on a decomposition of the state space into regions, or sets of states, but in a way that has nothing to do with the way Graphplan handles features.

For instance, *Sensory Graphplan* [21] aims at handling uncertainty about the initial state. In the given example in [21], there are two possible hypotheses for the initial state and therefore, *Sensory Graphplan* generates two separate plan graphs, from each initial state. Each

graph is related to a possible world. The plan search looks for a plan valid in both worlds. In this case, both worlds are characterized by simple features, but doing so, the authors have introduced some splitting in the search space, based on an partition of the state space at the root of the search space. We would like to authorize this at any stage of the search space building process, whenever introducing such distinctions may happen to prove useful.

2.5 Controlling the search space splitting

For the rest of this paper, one says that the search space building stage *splits* the search space, when it splits the current set of reachable states in several sub-sets, and then continues independently from each sub-set. This notion is explained in more details in [12], in terms of planning approaches and potential plan sets.

For example, an *FSS* (*Forward State Space*) search type of approach does *full splitting*. Indeed, as soon as an action is introduced in a plan prefix - narrowing down the current set of potential plans - the resulting set is pushed in a new branch of the search tree (see fig. 2). On the contrary, a (disjunctive) Graphplan-like approach does *no splitting at all* : all the possible actions are introduced together, and the set of all the potential plan sets they entail is considered as a whole when continuing planning. This is the basis of disjunctive planning.

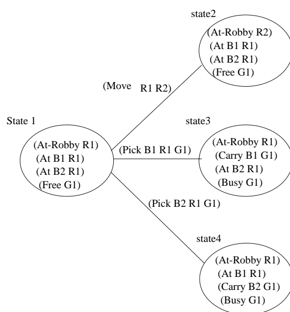


Figure 2. Here is the beginning of an FSS-search tree that would be developed for the gripper problem (with 2 balls and 1 gripper). Each node is a coherent state.

During the search space building stage, the state reachability may thus be more or less approximated. For example, Graphplan computes only pairwise mutex relations and therefore considers some states as reachable whereas they are not because of some three-wise constraint. In such a case, no plan can be found at backward search time, and a further extension of the search space is required.

The more splitting is used, the more accurate is this approximation. Doing full splitting insures exact reachability and optimality criteria computation. As a consequence, the checking stage in a deterministic goal oriented planning problem is trivial and classical Dynamic Programming tools can be applied directly if optimization has to be performed. With partial splitting, such criteria can at best be bounded, which may nevertheless prove useful.

The same way, the more splitting is used, the more narrow is the range of actions and propositions which are introduced. Indeed, we avoid some mistakes which may have led to the introduction of some actions that should be pruned. However, a higher level of splitting entails a higher level of duplication of work since each action may be introduced several times (from each subset of state) at each level. On the one hand, this makes the search stage easier - because part of the check is implicitly included in the classification, but on the other hand, this worsens the combinatorial aspects of the problem - because of the redundancy it entails.

2.6 Motivation for the use of Petri nets and tokens

The first benefits we expect from Graphplan-like schemes is the two-stage process of forward search space building followed by backward search. On the other hand, some information may be drawn during the building phase that we don't want to lose for the search (and optimization) phase. The intuitive idea is that token propagation via Petri-like graphs should allow to keep track of some of this information and reuse it later, either during the search space building phase or during the backward search phase.

The token propagation mechanisms should support the optimization capabilities which are barely compatible with Graphplan's disjunctive planning : as a first contribution, we show in the following that we can provide token propagation mechanisms that allow us to perform both *Forward State Space* search and disjunctive planning *"à la Graphplan"*.

Planning with Petri nets is not a completely new idea [9] : it is here combined with recent research on Graphplan [12, 11]. In [8, 19], the expected advantages of such a combination are discussed in terms of action and perception planning for autonomous agents. More generally, it allows to attach useful information concerning the planning process, uncertainty or risk to tokens and then track or use this information throughout the planning graphs.

3 Tokens and Petri nets

3.1 From a STRIPS domain to a Petri net

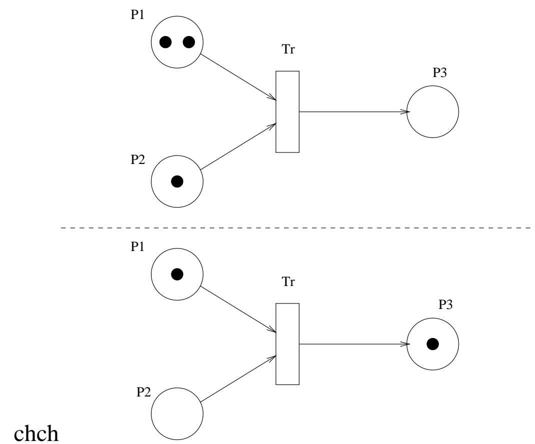


Figure 3. A simple Petri net, composed of a single transition Tr, and 3 places $P1$, $P2$, and $P3$. In the top figure, both $P1$ and $P2$ are marked by tokens (black dots), therefore the transition Tr can be triggered. On the bottom figure, the transition has been triggered : one token in each input place has been "consumed", and the output place $P3$ is marked by a token. Note that it remains a token in $P1$; however, since $P2$ is empty, the transition cannot be triggered. In more complex Petri nets, places can be connected to several transitions, as inputs or outputs.

A Petri net is composed of three types of elements : *places*, *transitions* and *tokens* (see fig.3). Places can be seen as token holders. They are connected to transitions as inputs or outputs. When a place contains one or more tokens, it is said to be *marked*. All of the places connected as input of a same transition must be marked before this transition can be triggered. When a transition is triggered, one token

of each of its input places is "consumed", and all of its output places are marked. This new marking can allow other transitions to be triggered and so on. In *colored* Petri nets, tokens can be of different colors. In this case, tokens'colors add new constraints for determining whether a transition can be triggered or not, and of course, additionnal rules allow transitions to compute the colors of tokens marking output places.

It is pretty straightforward to represent a STRIPS domain using Petri nets [9]. First, consider totally instantiated operators. For each possible proposition, the Petri net contains a place. Transitions correspond to operators. They are connected to the places related to their preconditions as inputs, and to the ones related to their effects as output. Once the STRIPS domain has been translated in a Petri net (see fig. 4), a state is represented by the global marking of the net.

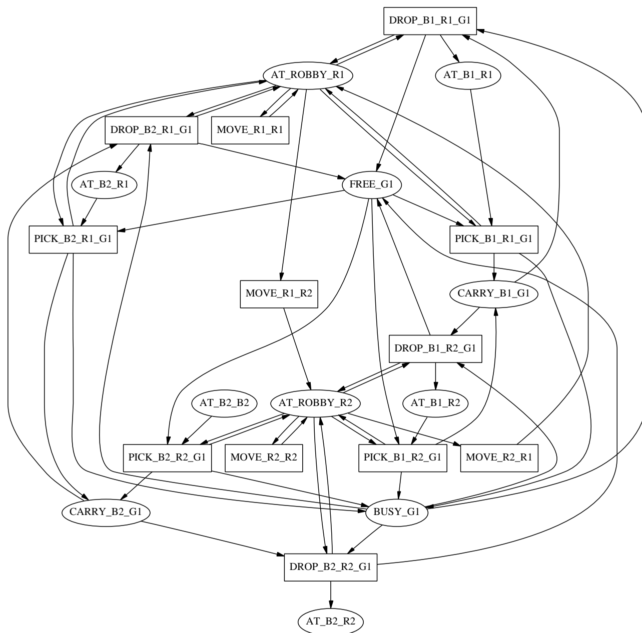


Figure 4. Petri net corresponding to an instance of a *gripper* domain. In this example, operators have been instantiated considering one gripper $G1$, two balls $B1$ and $B2$, and two rooms $R1$ and $R2$

When a place is marked by a token, the corresponding proposition is true (false otherwise, as in STRIPS). At a given time, the marking shows also all the transitions that can be triggered. This representation can be made more compact considering non-instantiated operators.

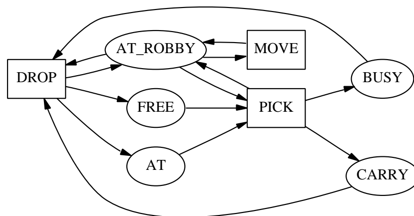


Figure 5. Petri net involving labels generated starting from a *gripper* domain (the same as the one of fig. 4). This graph, much more compact, represents non-instantiated operators. The pertinent instances will be brought using the tokens'labels.

3.2 From a PDDL domain to a colored Petri net

Let now each place correspond to a feature of the domain (see figure 5) : now each token has to hold a label on which is written an instantiation of the variables in the feature. From a propositional logic point of view, a place corresponds to a predicate, and the label of a token wears a particular binding. As a consequence, whether a transition can be triggered depends also on the labels of the tokens marking its preconditions. Reversely, the execution of a transition may not entail only a token motion but also some modification of their labels.

Practically, starting with a PDDL [18] description of a domain, its transcription into a Petri net proceeds automatically as follows. Each `:predicate` gives a place, and each `:action` a transition. Connecting to a transition places corresponding to preconditions or *positive effects* is obvious. *Negative effects* (i.e. when the operator destroys some of its preconditions) are naturally dealt with at the Petri net level, since a token marking a precondition place leaves it when the transition is triggered. Therefore, these bring no change to the net topology. A third type of effect has to be considered, we call them *implicit effects*. They encode the fact that some of the preconditions of the action remain true after its execution - i.e. propositions appearing in the precondition list without being in the *negative effect* list. These require additional links to be introduced in the network, in order to bring the tokens back to the places where they were before their transit through the transition (since these preconditions are still true). See fig. 6 for an example. [htbp]

(Pick ?B - ball ?R - room ?G - gripper)
Prec. : (At ?B ?R) (At-Robby ?R) (Free ?G)
Effects: (Carry ?B ?G)
~(At ?B ?R)

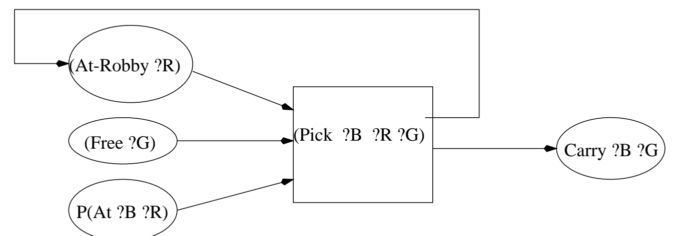


Figure 6. On top of the figure is the PDDL description of the operator $Pick$ (from the gripper domain). Below is its transcription as a Petri net transition. Note that there are two exiting links (on the right), whereas this operator has only one positive effect. This is because the top link - the one returning to At-Robby - is an implicit effect.

For each link in the Petri net, unification between variables of the proposition attached to the place and variables of the operator attached to the transition is completed and stored. This will make easier the computation related to the propagation of tokens through transitions (particularly when dealing with the modification of their labels).

Note that the Petri net obtained this way reflects exactly the domain as it is described in PDDL. Therefore, it is independent of any specific planning problem. Thus it needs to be built only once per domain.

4 Planning with tokens

4.1 Building the search space like Graphplan

Building the space of reachable states is done by propagating tokens from an initial marking, through transitions. The way Graphplan completes this kind of task (extending the planning graph) has proven its efficiency, and has displayed characteristics particularly attractive for planning [1, 13]. This section shows that the same can be done in our framework, inheriting by the way the same properties. Building the space of reachable states is done by propagating tokens from an initial marking, through transitions. Every place holds a list of token levels in which are recorded the successive markings. Therefore, a token is always considered within a given *token level* t of an item of the Petri net. It is linked to tokens in the $t-1$ *token level* of items of the net (showing where it comes from), and to tokens in the $t+1$ *token level* of items of the net (its potential destinations). This way, we can track the trajectory of a token during the propagation, and get the list of transitions it went through - i.e. the corresponding plan.

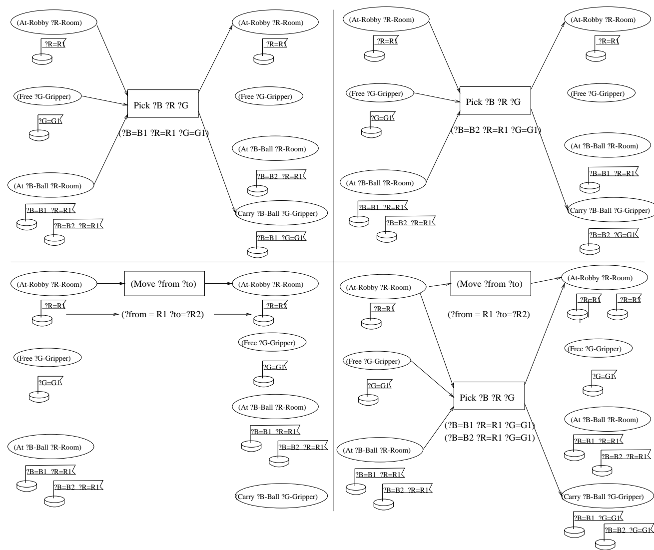


Figure 7. In every picture, the first column of places is the same, showing the marking at time t. The second column shows a possible marking at time t+1. The three first pictures correspond to the three transitions which can be triggered according to the current marking. The last one, on the bottom right hand corner, shows what we obtain with our approach : each possible transition has been triggered "virtually" so that the other ones could be triggered also.

At propagation time, as soon as the preconditions of a transition are marked, the transition is "virtually triggered". That is to say that even though tokens are sent to the next token level of the places corresponding to the positive and implicit effects of the transition, a copy of the current marking persists so that other possible propagations can be conducted (see fig. 7). As a consequence, at each step of the propagation, the global marking of the current token level corresponds to the union of all possible markings. Hence, it does not describe a single state but a set of reachable states.

As you can see in figure 8 the graph representing the tokens'connectivity is very similar to the one built by Graphplan.

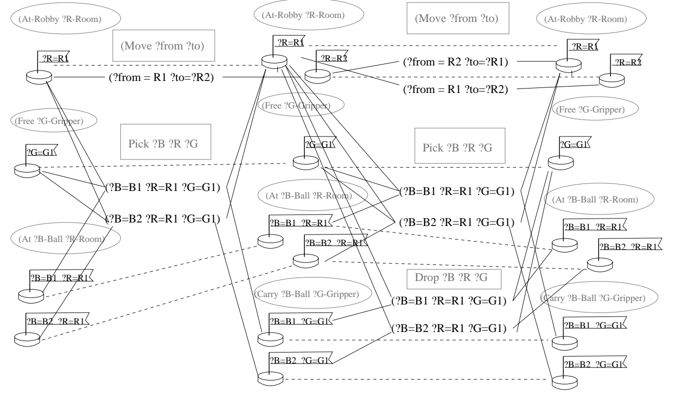


Figure 8. Each level shows the new token level of the places. In between, the transitions which caused the token motion are recorded. This plan graph is very much alike Graphplan's one.

So far, at each step, all the markings which can be deduced from the global marking of the net may include some unreachable ones. This is a common issue in disjunctive planning. In order to discard part of these impossible states, additional constraints are usually introduced : for instance, Graphplan computes *mutex relations*, and LCGP [5] computes *authorization relations*. In our system, mutex relations are computed, according to rules similar to Graphplan's ones (as stated in [1]). In terms of tokens, the *interference* rule (*"à la Graphplan"*) is as follows : if a transition T *"consumes"* a token, then it is mutex with all transitions using the same token (because T deleted one of their *preconditions*), and with all *other* transitions bringing an identical token (same label) in the same place (because T deleted one of their *positive effects*). In other words : two transitions cannot *"use"* the same token simultaneously, unless they both get and put back an identical token in a same place instantaneously (implicit effects). The *competing needs* rule (*"à la Graphplan"*) becomes : if there is a token t_1 triggering a transition, and a token t_2 triggering another transition, and if t_1 and t_2 are mutex, then these transitions are also mutex. Of course, the same thing could be done with *authorization* relations.

Similarly, among propositions : two tokens are mutex if every transition bringing the first one in its current place and giving it its current label is mutex with all the ones bringing the second one in its current place with its current label. Similarly to Graphplan, if two tokens marking precondition-places of a same transition are mutex, then the transition is not triggered.

The next section shows that some of the mutex relations need not to be explicitly computed, thanks to the use of colored tokens.

4.2 Building the search space slightly differently

The so far obtained planning graph is very similar to Graphplan's plan-graph, with the slight difference that each level is distributed among places and transitions. As a side effect, when searching for the particular instance of a proposition (to check whether an effect has already been introduced in the graph, for example), it is not necessary to search the whole list of propositions of the given level, but only the list of instances of the relevant place.

However, with the following rule, the use of colored tokens can allow to further avoid computing explicitly a number of *"permanent"*

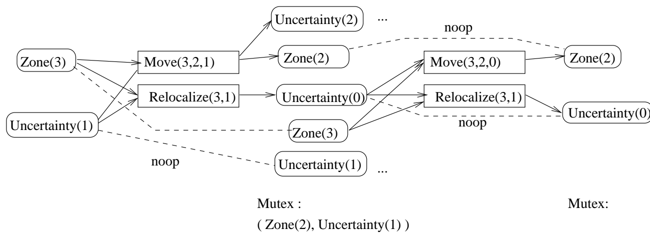


Figure 9. Example of mutex relation directly related to the planning problem. It disappears during graph construction. From the graph built for this mobile robotics problem, only a part relevant to our discussion is represented

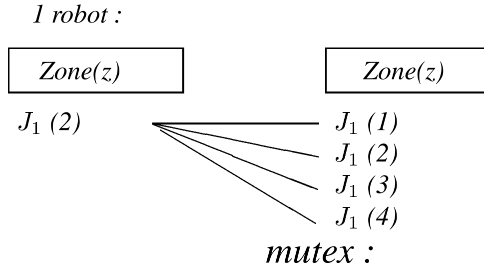


Figure 11. From zone 2, the robot can reach three other zones, or also remain where it is. Thanks to the use of colored tokens (here colors are represented by subscripts), no mutex needs to be expressed explicitly.

mutex relations between propositions carried by tokens of same color marking a same place (concurrent instantiations of a same feature) : *Two tokens of the same color (coming from the same past trajectory) marking a same place with two different labels are mutex.*

Indeed, two types of *mutex* relations are generated by Graphplan. Some of them are directly related to the planning problem itself, i.e. to the initial marking in the planning domain. They disappear after the plan-graph has been extended over some levels, and the range of reachable states consequently increased (see fig.9). Some other *mutex* relations, on the contrary, are *"permanent"*, or structural. Related to the domain itself, they will never disappear from the graph. They correspond to states which are strictly impossible, and therefore which can never be reached! Such *mutex* come from the fact that some features, or variables, cannot have more than one assignation at a given time, and in a given context. For example, a same robot cannot be concurrently in two different places (see fig. 10). In other words, some resources cannot be shared or some objects cannot be in different places at the same time. Comparable results are obtained by R.M. Simpson and T.L. McCluskey with their *Object-Graph planner* [20].

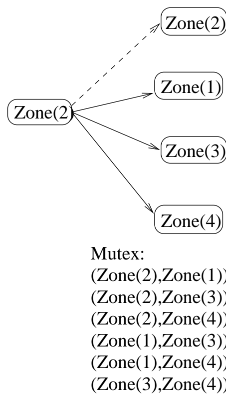


Figure 10. From zone 2, the mobile robot can reach 3 other zones, or also it can stay where it is. Graphplan is forced to introduce six permanent mutex relations.

Figure 11 shows the same situation as fig. 10 within our framework: No explicit mutex relation needs to be computed. They are all embedded in the token coloring.

As a more complete example, consider the possible motions of two robots *R1* and *R2* that can independently move in room *1*, *2*, *3* or *4*. Two tokens, denoted with different subscripts $_1$ and $_2$ are needed. Figure 12 shows the graph we get in this case. In the second level, place $InRoom(room, robot)$ is disjunctively marked by two sets of four tokens of same *"color"* (subscript $_1$ or $_2$ in this case). According to the rule, tokens of same subscript in the same level are all mutually exclusive with each other (a same robot cannot be simultaneously in different places!) but not with tokens of different subscript (the positions of the two robots are not dependent upon one another). There is a sort of parallel between the robot's non-ubiquity and the one of the token. In that sense, the tokens'colors are used here in order to represent the non-ubiquity of "objects" or "agents". In comparison to Graphplan, this rule leads to the computation of a smaller number of explicit *mutex* relations.

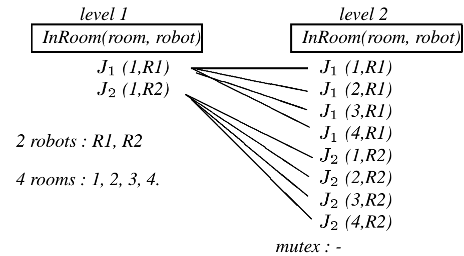


Figure 12. Both robots can reach three other rooms, or remain where they are. Thanks to token coloring, no mutex has to be expressed explicitly. Every position pair permitted by Graphplan is still permitted here.

The more numerous will be the variables not supporting multiple instantiations, and the literals which can be assigned to them, the more numerous will be the permanent *mutex*. Graphplan generate them explicitly, as regular *mutex* relations, making it more difficult to store and treat its data structures. We encode them implicitly through the use of colored tokens. These remarks may explain the speedup obtained in preliminary implementations and presented in table 1.

4.3 Handling colors - practically

Applying technically the above considerations is not always straightforward. Consider a transition with two preconditions and a single effect... which precondition token will give its color to the effect one - keeping in mind this will determine its potential mutex relations with other tokens marking the same place? Our concern being to

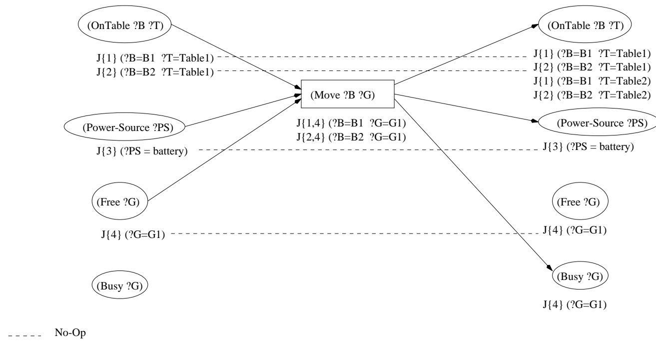


Figure 13. In the initial state, each token has a different color. As $Power - Source$ is an implicit effect, its color is not part of the action color (which therefore has only 2 components). In the second level, the third token of $OnTable$ wears the color 1 : this is the intersection between the action color ($\{1, 4\}$) and the set of colors marking this place in the initial state ($\{1, 2\}$). The token marking $Busy$ wears the color 4 because this is the only remaining component of the action color.

have a fully automatic planner, this choice must be made through a systematic method.

Firstly, here is a naive solution one could think of, based on the possible aggregation of colors. Let the colors be represented by sets of integers. In our example, say the first precondition is marked by a token of color C_1, and the second one by a token of color C_2, then the resulting color of the token marking the effect would be $C_1 \cup C_2$. For two tokens, of respective color C_i and C_j, marking a same place, if $C_i \cap C_j \neq \emptyset$ then there would be a mutex relation between both these tokens.

This does not work because the color treatment would not make any difference between tokens related to resources (sharable or reusable over time) and tokens related to objects (in the particular context of the transition being considered). Consider for example two tables with blocks on the first one, and two grippers used to move blocks from $Table1$ to $Table2$. The action $(MoveBlock?BGripper?G)$ requires the precondition $(OnTable?BTable_1)$ and $(Free?G)$. Of course, it is possible to move the block B_1 with the gripper G_1 and then to move the block B_2 with the same gripper G_1. This would cause the place $(OnTable?B?T)$ to be marked with two tokens, one carrying the label $(?B = B_1; ?T = Table2)$, and the other carrying the label $(?B = B_2; ?T = Table2)$. Unfortunately, both these tokens would have a color containing the color of the token marking $(FreeG_1)$, and therefore would be considered mutex in the "naive" plan graph whereas they are not. Here, the gripper plays the role of a resource, that is not sharable at one time, but usable several times in sequence.

Here is what is actually implemented in our system, which avoids these troubles. Let C_{action} be the union of the precondition colors of the considered action.

The color assigned to each effect is computed starting from C_{action}. In order to keep the tokens'coloring coherent with regard to the "objects" identification, some components of C_{action} have to be filtered out. Colors of tokens entering the transition (from preconditions) are related to some "objects" or "agents" ; the same consistency between "objects" and colors must be found among the tokens exiting the transition (towards effects). This is accomplished by maintaining a consistency between colors and places (instead of "objects" directly) and the best available reference concerning the coherent link between colors and places is the initial state. The assignment of effect colors is done as follows.

Preconditions being also implicit effects of the action are discarded from C_{action} (this takes care of sharable resources).

The effects of the action are split in two categories : places which were marked in the initial state (we know which colors are attached to them), and places which were not. For one of the first type, we keep from C_{action} only the components corresponding to the colors of tokens marking it in the initial state, and we remove these components from C_{action}. Effects of the second type will all receive the resulting C_{action}. Figure 13 shows this procedure on our simple example.

This approach makes a distinction, as much as possible, between resources and objects. The scenario we presented cannot occur, but still, the fact that the same ball cannot be on both tables at the same time remains encoded by the colors. Of course, in a description language like PDDL, as well as in natural language as a matter of fact, resources and objects using them can be confused sometimes - all the more as this distinction is highly context-dependent. For this reason, the number of mutex relations which will be encoded through the color manipulation depends upon both the way the domain is written and the initial state of the problem. Anyway, the planner remains complete since a lack of mutex relations is not harmful (only too many of them is harmful as far as it concerns completeness). Besides, regular explicit mutex checking is completed when token colors are not mutex, so that the backward search will have as many constraints as possible to be guided. Finally, note that in such a domain, colors of tokens marking places are single numbers - so that checking whether two tokens are mutex is easy and fast.

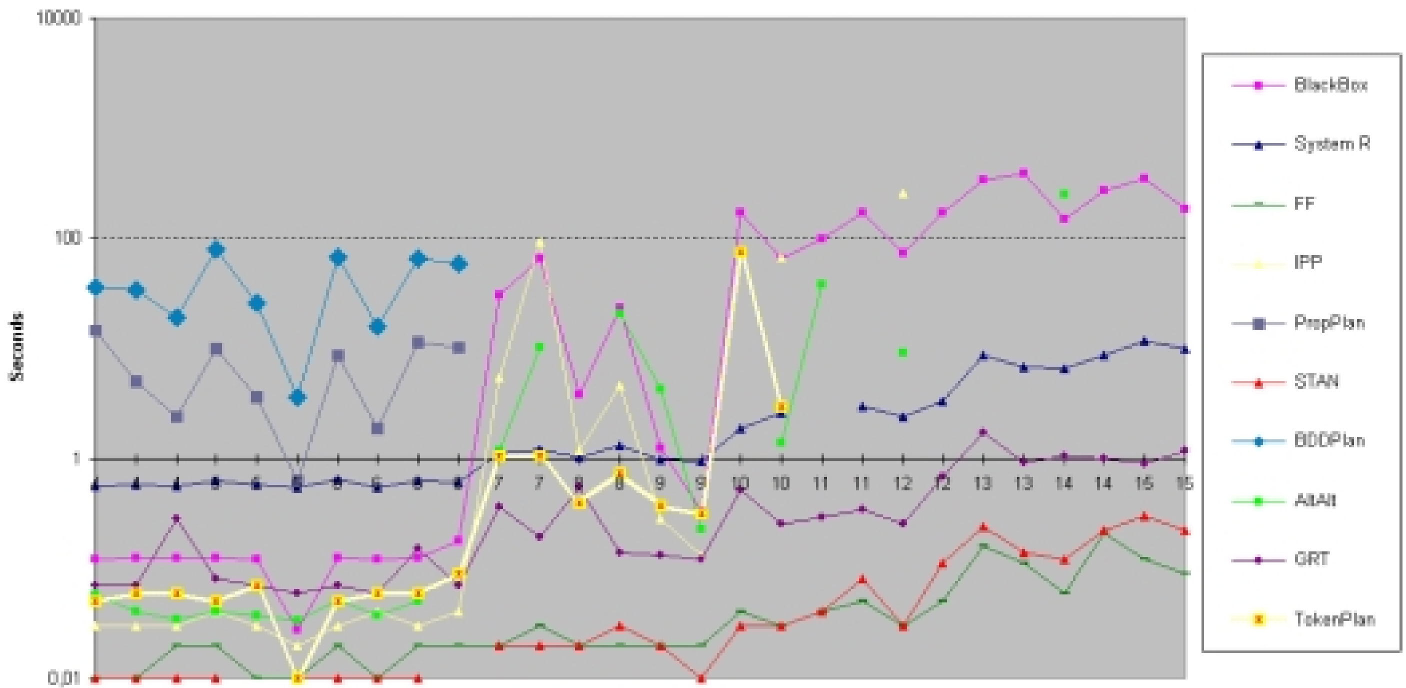
4.4 Obtaining a plan

Table 1. Comparison on the 6 balls GRIPPER domain with a code kindly provided by S. Kambhampati. Note that mutex relations not explicitly computed have been deducted thanks to tokens'colors.

	Graphplan + Csp tech.'s [11] 64 s 550 msec			Planning w/ tokens 4s 800 msec		
level	prop mut	action mut	records	prop mut	action mut	records
0	214	3858	0	136	2478	0
1	214	3858	1	136	2478	1
2	214	3858	45	136	2478	46
3	214	3858	254	136	2478	103
4	214	3858	1087	136	2478	204
5	214	3894	2724	136	2478	297
6	226	4206	5746	136	2478	781
7	250	4182	10774	136	2604	1261
8	322	3298	10237	148	1458	631
9	542	1466	0	112	680	0
10	220	176	0	140	121	0
11	0	0	0	0	0	0

As soon as the features of the *goals* are marked with the right tokens in the right places with the right labels, a backward search of the type of Graphplan's one can be applied to the graph generated by the token propagation in order to extract a valid plan - if any. A remarkable point is that the plan is output as a Petri net, so that all information about dependency relations among actions is preserved.

The presented planner based on token propagation mechanisms was implemented in Lisp on a Sun Sparc 10 Ultra and the GRIPPER domain has been automatically generated from a PDDL defini-



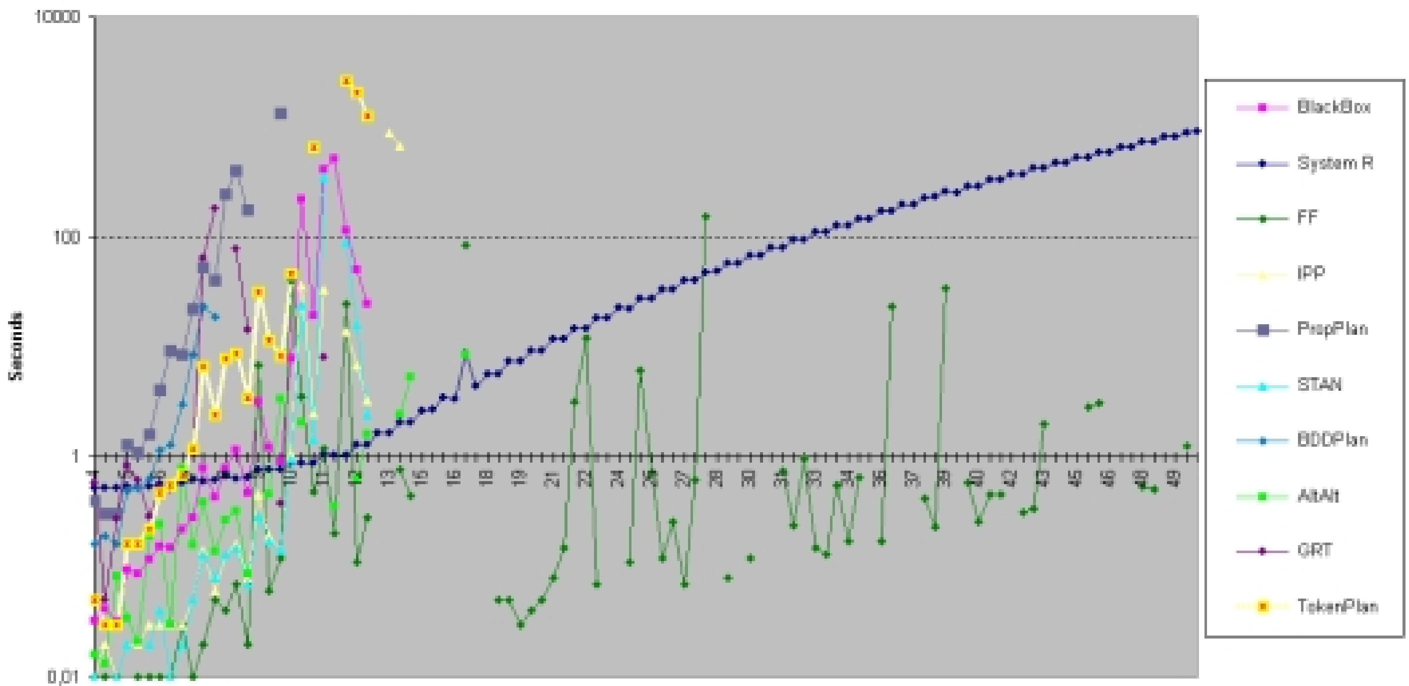


Figure 14. Both these graphs represent CPU-time performances of some of the planners participating to the AIPS'2000 planning competition. Missing points correspond to problems that could not be solved within the CPU-time limit (30 min).The top graph shows results on the Logistics domain, and the bottom one on the Blocks domain (abscissa values correspond there to the number of blocks involved in the problem)

tion file. A comparison was drawn with a Lisp code provided by S. Kambhampati.

The backward search algorithms are identical for both planners and are also from S. Kambhampati, who introduced in [11] the use of dynamic CSP techniques in order to boost the backward search phase. The table 1 gives both total planning time and for each level developed in the forward building phase, it compares the number of generated mutexes, among propositions or among actions, and sets of markings that are recorded as *"unreachable at that level"* after checking in the backward search phase.

The most remarkable feature in this comparison seems to be the difference in the number of mutex recorded by the two planners. The token planner need not record most of the *"permanent mutex"* classical Graphplan records and a subsequent simplification occurs both at propagation time, and backward search time : the number of constraints to check and the number of sets of markings to search are reduced consequently. There should be an increase in memory use, but it does not seem to be significant so far, contrary to the speedup.

Further experimentation has been completed during the AIPS'2000 Planning Competition. Our system entered it under the name *TokenPlan*. A sample of results is showed in figure 14, from data kindly provided by F. Bacchus, chair of this year competition. All the systems ran on a 500MHz Pentium III with 1GB of RAM. Some of them were implemented in C, C++, Lisp, or else in Java.

Despite these language differences, we can see TokenPlan is in the stream. Its performances are a bit better than BlackBox' (even though this one scales better), and roughly equivalent to IPP's. This shows the transcription of Graphplan in the Petri net framework is feasible and reasonable (no loss in performances is induced). The same way, as for other Graphplan based planner, it would be possible now to add features to Tokenplan in order to improve its performances (such as the computation of authorization relations instead of mutex relations [5] for instance).

4.5 Perspectives on search space splitting with token propagation

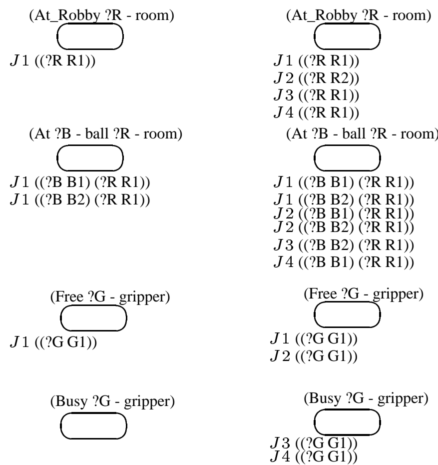


Figure 15. Here is the beginning of a full splitting search applied to the gripper problem in our framework. To make easier the comparison with fig. 2 we numbered the classes of tokens the same way we had numbered the state-nodes. Actions have been omitted for clarity.

Previous sections described how to build a fully disjunctive search space using token propagation. It can be rapidly checked that the same approach may as well generate full splitting by simply creating new classes of tokens each time a new state is reached : first consider tokens gathered in different classes (the class is represented here by an integer) - add the rule that in order to be triggered, a given transition must have all of its preconditions marked by tokens of the *same class* - add the rule that all the tokens marking the effects of a given transition are given a new class, which is not in use already. An example of the result is given in fig. 15. Some places may be marked by tokens of different classes wearing the same label. As an example, this is the case for $((?RR1))$ in fig. 15. This is the inherent redundancy of full splitting approaches.

As far as it concerns the example of [21], the splitting required to distinguish possible initial states can be achieved within our framework by introducing a different class of tokens for each possible world. Only one graph *per se* would be built.

Therefore, this may give the opportunity to duplicate only parts of the states that are actually different with respect to an optimization criterion : probability of reach and utility would be chief criteria for designing classes in a decision-theoretic planning problem such as the rocket one of figure 1.

Nevertheless, we have shown that we can provide token propagation mechanisms that allow us to perform both *Forward State Space* search and disjunctive planning *"à la Graphplan"*. Intermediate *"splitting"* strategies remain to be defined, implemented and optimized.

5 Conclusion

A lot of work remains to be done about the basic mechanisms of the proposed token-based planner and further validation and improvements are needed. Otherwise, it seems interesting to further study planning with intermediate levels of splitting as pointed out in [12, challenge4].

The basic idea is to be able to distinguish states or sets of states only when necessary. The framework proposed in this paper theoretically allows a large variety of splitting strategies, but the problem of controlling these strategies at planning time remains difficult. One solution would be to attach directly a sub-domain of the state space to the tokens. These sub-domains would correspond to a decomposition into regions of validity of properties related to utility functions or probability measures over the state space. Such a decomposition would be tailored for the computation of the stochastic optimality criteria just the same as regions of utility or reachability are computed geometrically in [7]. The advantage of tokens with that respect is that such a decomposition highly depends on the trajectory followed in the planning graph.

REFERENCES

[1] A. Blum and M. Furst, 'Fast planning through planning graph analysis', *AI*, (90), 281–300, (1997).

[2] A. Blum and J. Langford, 'Probabilistic planning in the graphplan framework', in *AIPS Workshop on 'Planning as Combinatorial Search'*, (1998).

[3] C. Boutilier, R.I. Brafman, and C. Geib, 'Structured reachability analysis for markov decision processes', in *UAI'98*, pp. 24–32, Madison, (jul. 1998).

[4] G. Boutilier, 'Decision-theoretic planning: Structural assumptions and computational leverage', *Journal of Artificial Intelligence Research*, **11**, 1–94, (1999).

[5] M. Cayrol, P. Regnier, and V. Vidal, 'New results about lcgp, a least committed graphplan', in *AIPS 2000*, pp. 273–282, Breckenridge, Colorado (USA), (2000).

[6] L. Dorst, M. van Lambalgen, and F. Voorbraak, eds. *Reasoning with Uncertainty in Robotics*. Springer, mar 1996.

[7] P. Fabiani and J.-C. Latombe, 'Dealing with geometric constraints in game-theoretic planning', in *IJCAI'99*, Stockholm, (aug. 1999).

[8] P. Fabiani and Y. Meiller, 'Théorie des jeux et planification pour le dilemme perception-action.', in *RFIA'2000*, PARIS, (2000). AFRIF-AFIA.

[9] F. Garcia, R. Mampey, and C. Barrouil, 'Génération de plans et révision des connaissances', in *RFIA'91*, Lyon-Villeurbanne, (novembre 1991).

[10] E. Guéré and R. Alami, 'Vers une planification en environnement dynamique', in *RFIA'2000*, PARIS, (2000). AFRIF-AFIA.

[11] S. Kambhampati, 'Planning graph as (dynamic) csp: Exploiting ebl, ddb and other csp techniques in graphplan', in *Ijcai'99*.

[12] S. Kambhampati, 'Challenges in bridging plan-synthesis paradigms', in *IJCAI'97*, (1997).

[13] S. Kambhampati, E. Lambrecht, and E. Parker, 'Understanding and extending graphplan', in *ECP'97*, (1997).

[14] S. M. Lavalle, *A Game-Theoretic Framework for Robot Motion Planning*, Electrical engineering, University of Illinois, Urbana-Champaign, 1995.

[15] M.L. Littman, T. Dean, and L.P. Kaelbling, 'On the complexity of solving markov decision processes', in *UAI'95*, pp. 394–402, Providence, (jul. 1995).

[16] L.J.Savage, *The Foundations of Statistics*, Dover, New York, 1972.

[17] R. D. Luce and H. Raiffa, *Games and Decisions*, John Wiley & Sons, New York, 1957.

[18] D. McDermott and AIPS-98 Planning Competition Committee, *PDDL -The Planning Domain Definition Language Version 1.2*, 1998.

[19] Y. Meiller and P. Fabiani, 'Perception-action dilemma at planning time : Getting the best out of game theory and classical planning', in *PLANSIG'99*, (1999).

[20] R.M. Simpson and T.L. McCluskey, 'An object-graph planning algorithm', in *PLANSIG'99*, (1999).

[21] D. S. Weld, A. R. Anderson, and D. E. Smith, 'Extending graphplan to handle uncertainty & sensing actions', in *AAAI'98*, (1998).