? reformuler: identifier comment un planner résoud un planning problem n'est pas vraiment une contribution:

This will lead us to identify how a planner solve a planning problem for any existing planning approaches.

4 General Planning Formalism

When designing intelligent systems, an important feature is the ability to make decisions and act accordingly. To act, one should plan ahead. This is why the field of automated planning is being actively researched in order to find efficient algorithms to find the best course of action in any given situation. The previous chapter allowed to lay the basis of knowledge representation. The way to represent the knowledge of planning domains is an important factor to take into account in order to conceive most planning algorithm.

present a new knowledge representation model

Automated planning really started being formally investigated after the creation of the Stanford Research Institute Problem Solver (STRIPS) by Fikes and Nilsson (1971). This is one of the most influential planners, not because of its algorithm but because of its formalism.

2 notions that define the domain

All planning formalisms are based on mainly two notions: *actions* and *states*. A state is a set of *fluents* that describe aspects of the world modeled by the domain. Each action has a logic formula over states that allows its correct execution. This requirement is called *precondition*. The mirror image of this notion is called possible *effects* which are logic formulas that are enforced on the current state after the action is executed. The domain is completed with a problem, most of the time specified in a separate file. The problem basically contains two states: the *initial* and *goal* states.

In this chapter, we will start with a popular planning problem as an example of what planning is about. Then, we will formalize general notions of planning using the functional formalism of chapter 2. This will lead to our first contribution in the form of identifying how a planner solve a planning problem for any existing planning approaches. From there, we can define a general planner that is able to solve all types of planning problems. To finish we will show how this formalism can be applied to every single planning paradigm.

relativiser car tu ne le montres/prouves pas:
« From there and thanks to the formalism we presented in chapter 2, we propose a general planning formalism intended to solve various types of planning problems »

4.1 Illustration

To illustrate how automated planners work, we introduce a typical planning problem called **block world**.

Example 29. In this example, a robotic grabbing arm tries to stack blocks on a table in a specific order. The arm is only capable of handling one block at a time. We suppose that the table is large enough so that all the blocks can be put on it without any stacks. Figure 4.1 illustrates the setup of this domain.

The possible actions are `pickup`, `putdown`, `stack` and `unstack`. There are at least three fluents needed:

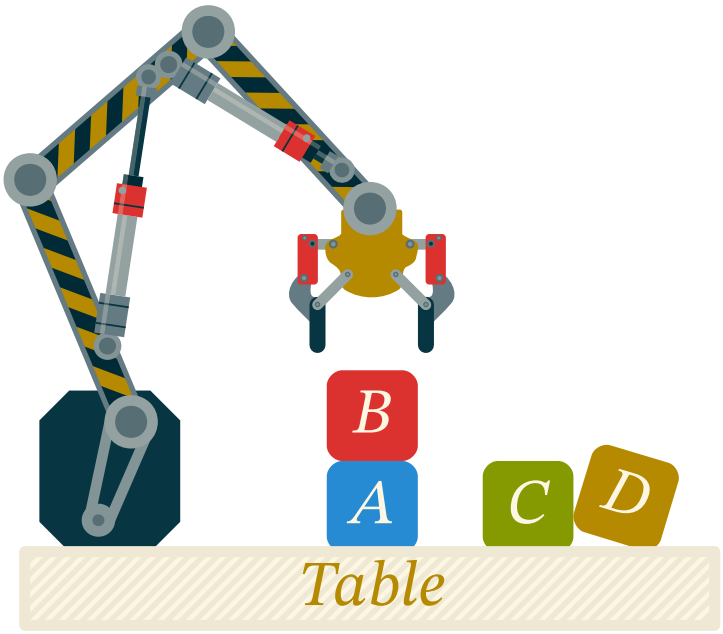


Figure 4.1: The block world domain setup.

- one to state if a given block is `down` on the table,
- one to specify which block is `held` at any moment and
- one to describe which block is stacked `on` which block.

We also need a special block to state when `noblock` is held or on top of another block. This block is a constant.

The knowledge we just described is called *planning domain*.

In that example, the initial state is described as stacks and a set of blocks directly on the table. The goal state is usually the specification of one or many stacks that must be present on the table. This part of the description is called *planning problem*.

In order to solve it we must find a valid sequence of actions called a *plan*. If this plan can be executed in the initial state and result in the goal state, it is called a *solution* of the planning problem. To be executed, each action must be done in a state satisfying its precondition and will alter that state according to its effects. A plan can be executed if all its actions can be executed in the sequence of the plan.

Example 30. For example, in the block world domain we can have an initial state with the $blockB$ on top of $blockA$ and the $blockC$ being on the table. In figure 4.2, we give a plan solution to the problem consisting of having the stack $\langle blockA, blockB, blockC \rangle$ from that initial state.

All automated planners aims to find such a solution to their planning problem. The main issue is that planning problem quickly become very expansive to solve. This is why planners often are evaluated on time and solution quality. The quality of a plan

becomeS

are often

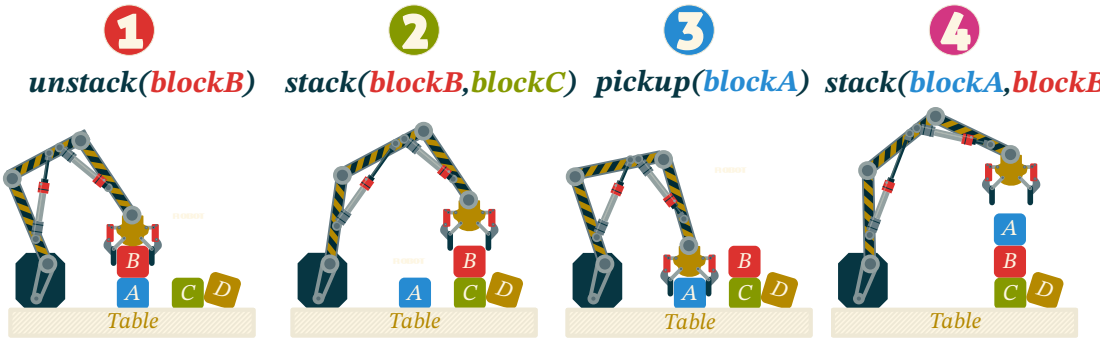


Figure 4.2: An example of a solution to a planning problem with a goal that requires three blocks stacked in alphabetical order.

is often measured by how hard it is to execute, whether by its execution time or by the resources needed to accomplish it. This metric is often called *cost* of a plan and is often simply the sum of the costs of its actions.

Automated planning is very diverse. A lot of paradigms shift the definition of the domain, actions and even plan to widely varying extents. This is the reason why making a general planning formalism was deemed so hard or even impossible:

> "*It would be unreasonable to assume there is one single compact and correct syntax for specifying all useful planning problems.*" Sanner (2010)

Indeed, the block world example domain we give is mostly theoretical since there is infinitely more subtlety into this problem such as mechatronic engineering, balancing issues and partial ability to observe the environment and predict its evolution as well as failure in the execution. In our example, we didn't mention the misplaced $blockD$ that could very well interfere with any execution in unpredictable ways. This is why so many planning paradigms exist and why they are all so diverse: they try to address an infinitely complex problem, one sub-problem at a time. In doing so we lose the general view of the problem and by simply stating that this is the only way to resolve it we close ourselves to other approaches that can become successful. Like once said:

> "*The easiest way to solve a problem is to deny it exists.*" Asimov (1973)

However, in the next section we aim to define such a general planning formalism. The main goal is to provide the automated planning community with a general unifying framework.

4.2 Formalism

In this section, we will present how automated planning works by using the formalism we first described in chapter 2. This leads to a general formalism of automated planning. The goal is to explain what is planning and how it works. First we must express the knowledge domain formalism, then we describe how problems are represented and lastly how a general planning algorithm can be envisioned.

4.2.1 Planning domain

In order to conceive a general formalism for planning domains, we base its definition on the formalism of SELF. This means that all parts of the domain must be a member of the universe of discourse $\mathbb{U}$.

4.2.1.1 Fluents

First, we need to define the smallest unit of knowledge in planning, the fluents.

Definition 28 (Fluent)**.** A planning fluent is a predicate $f \in F$.

Fluents are signed. Negative fluents are noted $\neg f$ and behave as a logical complement. We do not use the closed world hypothesis: fluents are only satisfied when another compatible fluent is provided.

The name "fluent" comes from their fluctuating value. Indeed the truth value of a fluent is meant to vary with time and specifically by acting on it. In this formalism we represent fluents using either parameterized entities or using statements for binary fluents.

Example 31. In our example, back in section 4.1, we have three predicates: `down`, `held` and `on`. They can form many fluents like $held(no - block)$, $on(blockA, blockB)$ or $\neg down(blockA)$. When expressing a fluent we suppose its truth value is $\top$ and denote falsehood using the negation $\neg$.

4.2.1.2 States

When expressing states, we need a formalism to express sets of fluents as formulas.

Definition 29 (State)**.** A state is a logical formula of fluents. Since all logical formulas can be reduced to a simple form using only $\wedge$, $\vee$, and $\neg$, we can represent states as *and/or trees*. This means that the leaves are fluents and the other nodes are states. We note states using small squares $\square$ as it is often the symbol used in the representation of automates and grafcets.

Example 32. In the domain block world, we can express a couple of states as :

- $\square_1 = held(noblock) \wedge on(blockA, blockB) \wedge down(blockC)$
- $\square_2 = held(blockC) \wedge down(blockA) \wedge down(blockB)$

In such a case, both state $\square_1$ and $\square_2$ have their truth value being the conjunction of all their fluents. We can express a disjunction in the following way: $\square_3 = \square_1 \vee \square_2$. In that case $\square_3$ is the root of the and/or tree and all its direct children are or vertices. The states $\square_1$ and $\square_2$ have their children as *and vertices*. All the leaves are fluents. This tree is presented in the figure 4.3.

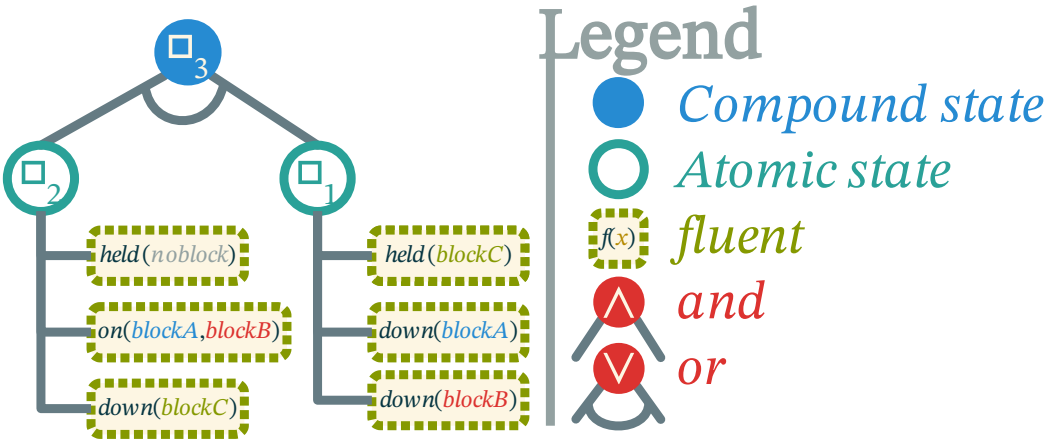
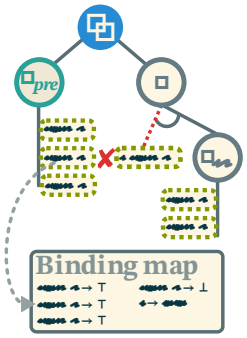


Figure 4.3: Example of a state encoded as an and/or tree.

4.2.1.3 Verification and binding constraints



toujours la même remarque: si tu laisses l'image autant la mettre en grand et faire en sorte que cela illustre ton texte et example.

Je pense qu'il faut la garder mais la mettre en grand, c'est important que la planif classique soit bien comprise car dans ton jury il n'y a pas que des gens issus de la planif !

When planning, there are two operations that are usually done on states: verify if a precondition fits a given state and then apply the effects of an action.

Classical planning have a clear distinction between preconditions and effects. Preconditions are predicates that needs to be satisfied before the action can be executed. In classical formalisms, effects are separated into positive and negative effects (Ghallab *et al.* 2004). Most of the time, planners will only support simple positive and grounded effects in order to make planning much easier.

In our model we consider preconditions and effects as states. The idea is to make the planning formalism as uniform as possible in order to factor most operations into generic ones.

The verification is the operation $\square_{pre} \vDash \square$ that has either no value when the verification fails or a binding map for variables and fluents with their respective values.

The algorithm is a regular and/or tree exploration and evaluation applied on the state $\boxplus = \square_{pre} \wedge \square$. During the evaluation, if an inconsistency is found then the algorithm returns nothing. Otherwise, at each node of the tree, the algorithm will populate the binding map and verify if the truth value of the node holds under those constraints. All quantified variables are also registered in the binding map to enforce coherence in the root state. If the node is a state, the algorithm recursively applies until it reaches fluents. Once $\boxplus$ is valuated as true, the binding map is returned.

Example 33. Using previously defined example states $\square_{1,2,3}$, and adding the following:

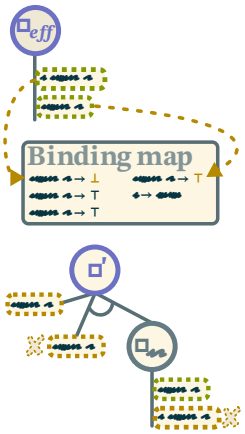
- $\square_4(x) = \{held(noblock), down(x)\}$ and
- $\square_5(y) = \{held(y), \neg down(y)\}$,

We can express a few examples of fluent verification:

- $held(noblock) \vDash held(x) = \{x = noblock\}$
- $\neg held(x) \vDash held(x) = \varnothing$

4.2.2 Effect Application

IDEM TROP PETIT !



Once the verification is done, the binding map is kept until the planner needs to apply the action to the state. In our formalism, effects *are* states instead of describing what fluents are added or deleted. The application will therefore be much different than with classical planning. Indeed the goal of the application will be to envorce the validity of the effect state while maintaining the coherence of the next state.

enForce

The application of an effect state is noted $\square_{eff}(\square) = \square'$ and is very similar to the verification. The algorithm will traverse the state $\square$ and use the binding map to force the values inside it. The binding map is previously completed using $\square_{eff}$ to enforce the application of its new value in the current state. This leads to changing the state $\square$ progressively into $\square'$ and the application algorithm will return this state.

4.2.2.1 Actions

Actions are the main mechanism behind automated planning, they describe what can be done and how it can be done.

Definition 30 (Action)**.** An action is a parameterized tuple $a(args) = \langle pre, eff, \gamma, ¢, d, \mathbb{P}, \mathbb{M} \rangle$ where:

- *pre* and *eff* are states that are respectively the **preconditions and the effects** of the action.
- γ is the state representing the **constraints**.
- $¢$ is the intrinsic **cost** of the action.
- d is the intrinsic **duration** of the action.
- $\mathbb{P}$ is the prior **probability** of the action succeeding.
- $\mathbb{M}$ is a set of **methods** that decompose the action into smaller simpler ones.

Operators take many names in different planning paradigms: actions, steps, tasks, etc. In our case we call operators, all fully lifted actions and actions are all the possible instances (including operators).

In order to be more generalist, we allow in the constraints description, any time constraints, equalities or inequalities, as well as probabilistic distributions. These constraints can also express derived predicates. It is even possible to place arbitrary constraints on order and selection of actions.

Actions are often represented as state operators that can be applied in a given state to alter it. The application of actions is done by using the action as a relation on the set of states $a : \square \to \square$ defined as follows:

$$a(\square) = \begin{cases} \varnothing, & \text{if } pre \nvDash \square = \varnothing \\ eff(\square), & \text{using the binding map otherwise} \end{cases}$$

Example 34. A useful action we can define from previously defined states is the following:

$$pickup(x) = \langle \square_4(x), \square_5(x), (x : Block), 1.0¢, 3.5s, 75\%, \varnothing \rangle$$

That action can pick up a block x in 3.5 seconds using a cost of 1.0 with a prior success probability of 75%.

4.2.2.2 Domain

The planning domain specifies the allowed operators that can be used to plan and all the fluents they use as preconditions and effects.

Definition 31 (Domain)**.** A planning domain $\mathcal{D}$ is a set of **operators** which are fully lifted *actions*, along with all the relations and entities needed to describe their preconditions and effects.

Example 35. In the previous examples the domain was named block world. It consists in four actions: $pickup, putdown, stack$ and $unstack$. Usually the domain is self contained, meaning that all fluents, types, constants and operators are contained in it.

4.2.3 Planning problem

The aim of an automated planner is to find a plan to satisfy the goal. This plan can be of multiple forms, and there can even be multiple plans that meet the demand of the problem.

4.2.3.1 Solution to Planning Problems

Definition 32 (Partial Plan / Method)**.** A partially ordered plan is an *acyclic* directed graph $\pi = (A_\pi, E)$, with:

- A_π the set of **steps** of the plan as vertices. A step is an action belonging in the plan. A_π must contain an initial step a^0_π and goal step a^*_π as convenience for certain planning paradigms.
- E the set of **causal links** of the plan as edges. We note $l = a_s \xrightarrow{\square} a_t$ the link between its source a_s and its target a_t caused by the set of fluents $\square$. If $\square = \varnothing$ then the link is used as an ordering constraint.

With this definition, any kind of plan can be expressed. This includes temporal, fully or partially ordered plans or even hierarchical plans (using the methods of the actions Π). It can even express diverse planning results.

reminiScent

The notation can be reminicent of functional affectation and it is on purpose. Indeed, those links can be seen as relations that only affect their source to their target and the plan is a graph with its adjacence function being the combination of all links.

donner des exemples pour illustrer, pour qu'on voit que tous types de plan peut être exprimé avec ton formalisme (ce n'est pas évident et ce n'est pas au lecteur de faire cet effort, il ne te croira pas juste parce que tu le dis)

In our framework, *ordering constraints* are defined as the transitive cover of causal links over the set of steps. We note ordering constraints: $a_a \succ a_s$, with a_a being *anterior* to its *successor* a_s. Ordering constraints cannot form cycles, meaning that the steps must be different and that the successor cannot also be anterior to its anterior steps: $a_a \neq a_s \wedge a_s \not\succ a_a$. If we need to enforce order, we simply add a link without specifying a cause. The use of graphs and implicit order constraints helps to simplify the model while maintaining its properties. Totally ordered plans are made by specifying links between all successive actions of the sequence.

Example 36. In the section 4.1, we described a classical fully ordered plan, illustrated in figure 4.2. A partially ordered plan has a tree-like structure except that it also meets in a "sink" vertex (goal step). We explicit this structure in figure 4.4. This figure is an example of how partially ordered plans are structured. The main feature of such a graph is its acyclicity.

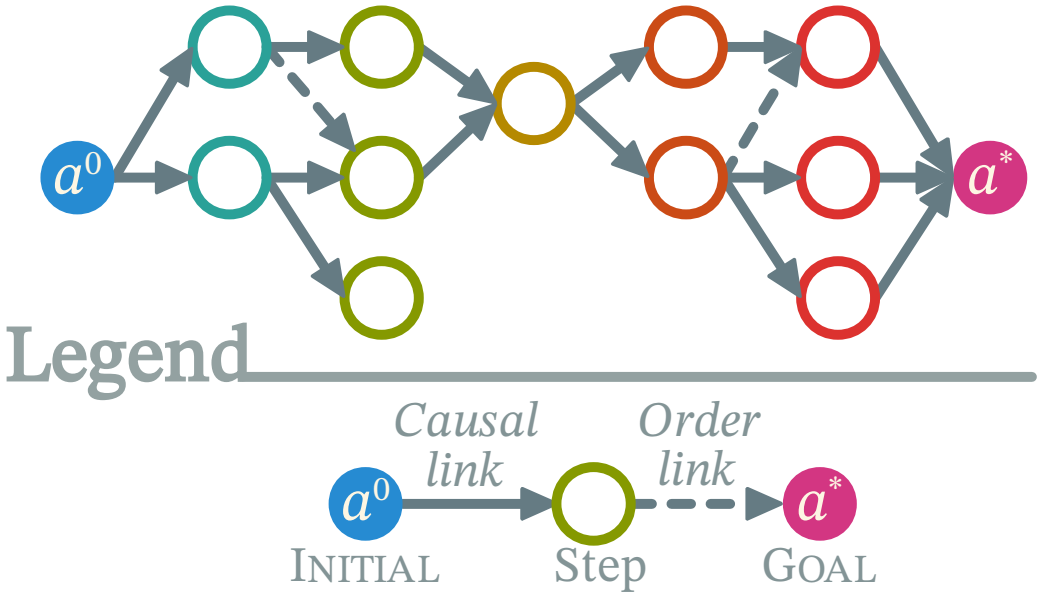


Figure 4.4: Structure of a partially ordered plan.

4.2.3.2 Planning Problem

With this formalism, the problem is very simplified but still general.

Definition 33 (Planning Problem)**.** The planning problem is defined as the **root operator** ω which methods are potential solutions of the problem. Its preconditions and effects are respectively used as initial state and goal description.

As for the preconditions and effects, we decided to factorize processes by making the homogenEous problem homogenous. Indeed, since the problem itself is an action and that each action also can have methods comprised of actions, it is possible to treat problems and actions the same way (especially useful for hierarchical planning). Most of the specific

pas clair

notions of this framework are optional. Any planner using it will define what features it supports when compiling input domains and problems.

Since we base our planning formalism on the one of SELF, we can express the structure of our formalism using the same extended Venn diagram as in figure 3.6. The figure 4.5 explains how planning uses the previously defined relations to make its structure homogenous and contained within the universe $\mathbb{U}$. We can also see the relation $\aleph$ between the set of all states $\square$ and the set of all fluents F that forms the previously shown and-or trees. Multiple relations are used for actions in A. Those are all explained in definition 30. We included the root operator ω in the set of all actions. It is interesting to note how the set of plans $\mathbb{\Pi}$ is by definition a set of graphs since it uses a connectivity function χ between actions and states.

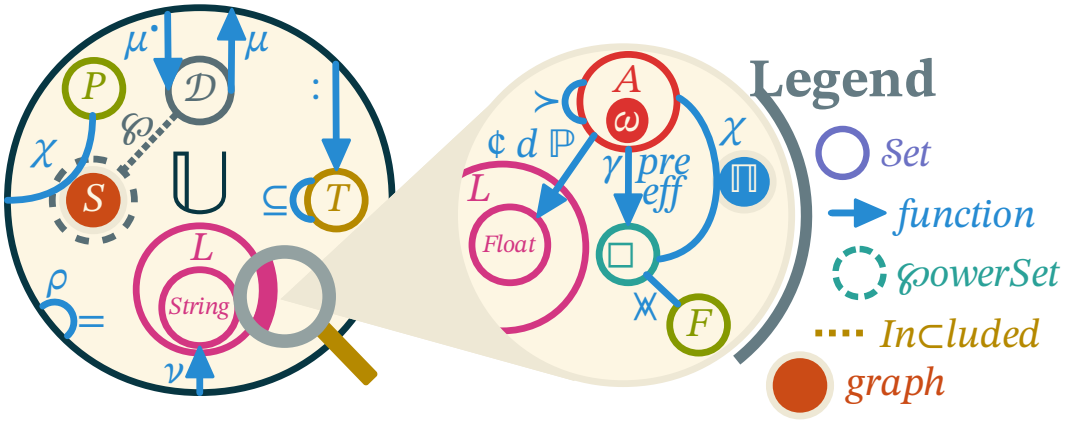


Figure 4.5: Venn diagram extended from the one from SELF to add all planning knowledge representation.

4.3 Planning search

A general planning algorithm can be described as a guided exploration of a search space. The detailed structure of the search space as well as search iterators is dependent on the planning paradigm used and therefore are parameters of the algorithm. There are also quite a few aspects that need consideration like solution and temporal constraints.

4.3.1 Search space

Definition 34 (Planner). A planning algorithm, often called planner, is an exploration of a search space $\mathbb{S}$ partially ordered by an iterator $\chi_{\mathbb{S}}$ guided by a heuristic h. From any problem $\mathbb{p}$ every planner can derive two pieces of information immediately:

- the starting point $\mathbb{s}_0 \in \mathbb{S}$ and
- the solution predicate $?_{\mathbb{S}^*}$ that gives the validity of any potential solution in the search space.

Formally the problem can be viewed as a path-finding problem in the directed graph $g_{\mathbb{S}}$ formed by the vertex set $\mathbb{S}$ and the adjacency function $\chi_{\mathbb{S}}$. The set of solutions is therefore expressed as:

$$\mathbb{S}^* = \left\{ \mathbb{s}^* : \langle \mathbb{s}_0, \mathbb{s}^* \rangle \in \chi_{\mathbb{S}}^+(\mathbb{s}_0) \wedge ?_{\mathbb{s}^*} \right\}$$

We note a provided heuristic $h(\mathbb{s})$. It gives off the shortest predicted distance to any point of the solution space. The exploration is guided by it by minimizing its value.

In order to accomodate some specific planning paradigm, it is interesting to have a common way to encode search constraints such as the expected number of solutions, or even the allocated time to complete the search.

4.3.2 Solution constraints

As finding a plan is computationally expensive, it is sometimes better to try to find either a more generally applicable plan or a set of alternatives. This is especially important in the case of execution monitoring or human interactions as proposing several relevant solutions to pick from is a very interesting feature. Also, in planning time is of the essence and it is useful to inform the planner of the time it has to find a solution.

reformuler, je ne comprends pas

We note $\gamma_{\mathbb{S}}$ the **set of constraints on the solution**. These constraints are exprimed like states: as a set of predicates that evaluates to true once the solution is meeting the expectations.

These constraints can vary widly between planning paradigm. As such, it is interesting to standardize how each planning paradigm will encode extra problem specification and requirements. In order to explain why solution contraints are relevant to planning, we will explain it all on a simple example.

A

Example 37. Bob is ~~an~~ machinist and plans to make a part using a mill. This part is needed for a bigger project and must be ready by a specific deadline.

aN

Figure 4.6 shows a unfortunate accident. While taking a tea break, Bob spills over his drink over the mill's electronics making it unusable. Bob originally planned to make the part using computer control. This is a problem because Bob needs an alternative plan now that his original plan failed.

In the following sections we will explain what Bob can do to meet his deadline and how it is relevant for planning.

4.3.2.1 Diversity

A planning paradigm in particular requires explicit constraints on the solution. This paradigm is called *diverse planning*. The goal is to find several plans that are all solution but all different enough as to be different approaches. It can be thought as a form of hierarchical planning where the composition isn't specified but we expect nonetheless different methods.

je ne vois pas du tout l'intérêt de cette image !

Figure 4.6: Bob regreting life choices and thinking about what to do next

cette notion existe en diverse planning ? Si oui, le dire et citer (même si c'est différent

In order to quantify the expected diversity of a set of solution we introduce to $\gamma_{\mathbb{S}}$ the plan deviation metric Δ that allows to compare how much two plans are different.

Example 38. Bob could have come with two different plans to make that part such as using the mill manually or using a subcontractor to make the part.

The main idea behind diverse planning is to come prepared and to present very different plans. For example Bob couldn't just have used a plan that would have consisted of switching the ~~or~~rientation of the work on the mil or anything too similar to the original plan.

millL

4.3.2.2 Cardinality

In diverse planning, another criteria that is user specified is the cardinality of the set of solution. We note k the number of expected different solutions. This simply makes the process return when either it found k solutions or when it determined that $k > |\mathbb{S}^*|$.

In classical planning, $k = 1$ since only the best plan is expected. This parameter is added to the set of constraints $\gamma_{\mathbb{S}}$.

Example 39. In our example, Bob clearly came prepared with a unique plan. The re-planning he is now doing could have been avoided with more plans prepared in advance.

In practice, finding a good amount of plans is quite arbitrary and context dependant and so often left at the user's discretion.

4.3.2.3 Probability

For probabilistic planning, all elements of the probability distributions used are typically included in the domain. The prior probability distribution noted $\mathbb{P}$ is therefore encoded into $\gamma_{\mathbb{S}}$.

Probabilistic uses this probability distribution to compute a *policy* that helps reach a goal. A policy is a heuristic guide that returns the action most likely to succeed in any given situation. So a policy is basically a function $\mathbb{pol} = \square \to \{\!|\max \circ \mathbb{P}_{\square} : A|\!\}$ that will always give the action with the maximum success probability knowing the current state.

Using the mapping notation $\{\!|\}$ from chapter 2.

Example 40. Bob can use a policy to solve that issue. In his case it is probably more of a "protocol" or a document instructing of what to do in most situations. That way, Bob doesn't have to improvise and will follow what was decided before.

4.3.2.4 Temporality

Another aspect of planning lies in its timing. Indeed sometimes acting needs to be done before a deadline and planning are useful only during a finite timeframe. This is done even in optimal planning as researchers evaluating algorithms often need to set a timeout in order to be able to complete a study in a reasonable amount of time. Indeed, often in efficiency graphs, planning instances are stopped after a defined amount of time.

This time component is quite important as it often determines the planning paradigm used. It is expressed as two parameters:

- $t_{\mathbb{S}}$ the allotted time for the algorithm to find at least a fitting solution.
- t^* additional time for plan optimization.

This means that if the planner cannot find a fitting solution in time it will either return a timeout error or a partial or abstract solution that needs to be refined. Anytime planners will also use these parameters to optimize the solution some more. If the amount of time is either unknown or unrestricted the parameters can be omitted and their value will be set to infinity.

Example 41. In our running example, Bob has a specified deadline to abide with. This kind of constraints are often implicit and not given to the planner. Classical planner will simply do a best effort to find the best solution and are only evaluated up to a certain time to compare results between approaches.

4.4 General planner

A general planner is an algorithm capable to resolve all types of planning domains and problems. This means that it becomes possible to compare the characteristics of

je ne comprends pas : quelles autres approches on peut alors utiliser pour comparer les planners si on en a plus qu'un ???

planners with completely different approaches. It also makes it possible to use several planning paradigms at once (e.g. durative actions and probabilistic planning combined). The formalization of such a planner allows for a more general explannation and description of planning problems and solutions.

4.4.1 Formalization

rappeler à quoi correspond chaque notation gs, s0, … pour éviter que le lecteur aille rrechercher dans le texte précédent

We note a general planner $\Pi^*(g_{\mathbb{S}}, \mathbb{s}_0, ?_{\mathbb{s}^*}, h, \gamma_{\mathbb{S}}, \mathcal{D})$ an algorithm that can find solutions using any valid instance of the planning formalism. si on choisit une instance elle sera capable de résoudre tous les formalismes ???

The most important point to understand is that a general planner is a shortest path algorithm. These kind of algorithms are very common in AI and are often used as example of classical algorithms. One of the first classical shortest path algorithm is called $A*$ (Hart *et al.* 1968). It is a simple heuristic powered shortest path algorithm.

pas très clair: A^* et K^* peuvent représenter une seul et même instance de ton general planner ?

A variant of the $A*$ algorithm can be used for diverse planning. This algorithm is called $K*$ after the term k used to denote the number of expected solutions (Aljazzar and Leue 2011, alg. 1).

In automated planning, this last algorithm is used as the base of some diverse planners (Stentz and others 1995; Riabov *et al.* 2014). Using our formalism, the parameters are as follows: $K^*(g_{\mathbb{S}}, \mathbb{s}_0, ?_{\mathbb{s}^*}, h)$. In this case, the solution predicate contains the solution constraints $\gamma_{\mathbb{S}}$.

Of course this algorithm is merely an instance of a general planner algorithm. The algorithm has been chosen to be general and its efficiency hasn't been experimentally tested.

tu parles de quel algo K^* ou d'autre chose ? ce n'est pas clair ici ce que tu veux dire et ce dont tu parles. « the algorithm has been chosen to be general … » c'est A^* , le general planner ???

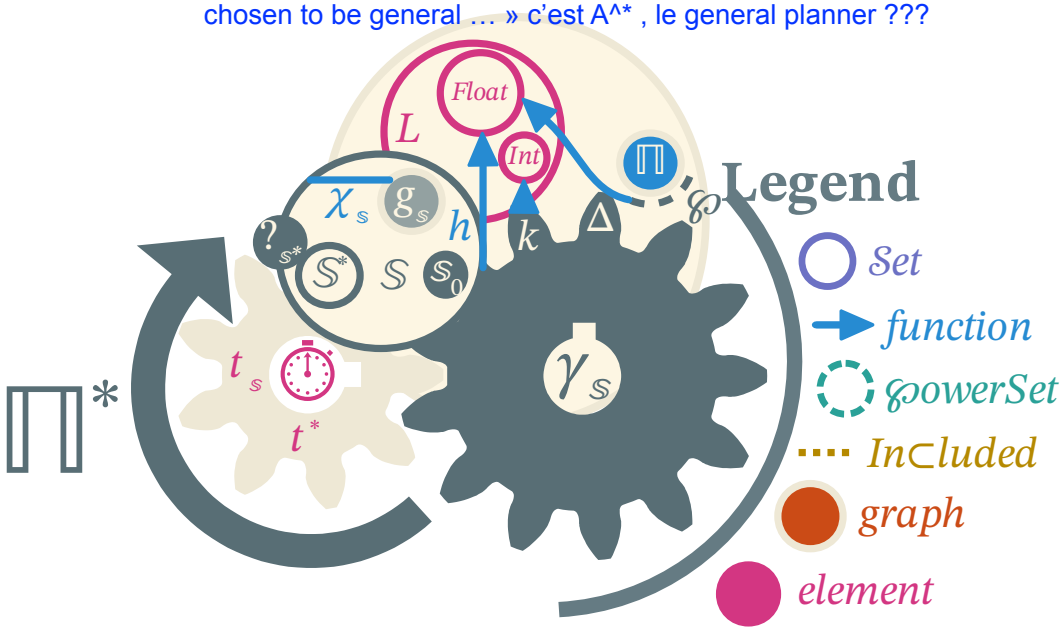


Figure 4.7: Venn diagram extended with general planning formalism.

Figure 4.7 is an illustration of how such a general planner is structured. The darker inner circle represents the search space $\mathbb{S}$ as defined in definition 34. The gears represents different aspects of the solution constraints $\gamma_{\mathbb{S}}$.

Now that a general planner has been formed it is quite relevant to apply it on every planning paradigms to show that it can fit them all.

tu ne l'appliques pas dans la suite, tu montres quels sont ces différents paramètres dans chq paradigme : modifier le terme apply, sinon on s'attend à ce que tu le testes/expérimente

4.5 Classical Planning Paradigms

One of the most comprehensive work on summarizing the automated planning domain was done by Ghallab *et al.* (2004). This book explains the different planning paradigm of its time and gives formal description of some of them. This work has been updated later (Ghallab *et al.* 2016) to reflect the changes occurring in the planning community.

In the following sections we will show instances of the general planner for each planning formalism. We will omit the last three parameters $h, \gamma_{\mathbb{S}}, \mathcal{D}$. We do so in order to use the partial application of chapter 2 and obtain a specific planner ready to be fully instanciated by the user (the domain is built in a way so that it doesn't depends on the paradigm).

4.5.1 State-transition planning

The most classical representation of automated planning is using the state transition approach: actions are operators on the set of states and a plan is a finite-state automaton. We can also see any planning problem as either a graph exploration problem or even a constraint satisfaction problem. In any way that problem is isomorph to its original formulation and most efficient algorithms use a derivative of A* exploration techniques on the state space.

The parameters for state space planning are trivial:

$$\mathbb{\Pi}^*_{\square} = \mathbb{\Pi}^* \left((\square, A), \mathit{pre}(\omega), \mathit{eff}(\omega) \right)$$

This formulation takes advantage of several tools previously described. It uses the partial application of a function to omit the last parameters. It also defines the search graph using the set of all states $\square$ as the vertices set and the set of all available actions A as the set of edges while considering actions as relations that can be applied to states to make the search progress toward an eventual solution. We also use the binary nature of states to use the effects of the root operator as the solution predicate.

Usually, we would set both $t_{\mathbb{S}}$ and t^* to an infinite amount as it is often the case for such planners. These parameters are left to the user of the planner.

State based planning usually supposes total knowledge of the state space and action behavior. No concurrence or time constraints are expressed and the state and action

space must be finite as well as the resulting state graph. This process is also deterministic and doesn't allow uncertainty. The result of such planning is a totally ordered sequence of actions called a plan. The total order needs to be enforced even if it is unnecessary.

All those features are important in practice and lead to other planning paradigms that are more complex than classical state-based planning.

4.5.2 Plan space planning

Plan Space Planning (PSP) is a form of planning that uses plan space as its search space. It starts with an empty plan and tries to iteratively refine that plan into a solution.

The transformation into a general planner is more complicated than for state-based planning as the progress is made through refinements. We note the set of possible refinements of a given plan $r = \mathbb{π} \rightarrow \{\!|\odot : \otimes(\mathbb{π})|\!\}$ with $\otimes$ being the flaws and $\odot$ the resolvers. Each refinement is a new plan in which we fixed a *flaw* using one of the possible *resolvers* (see definition 35 and definition 36).

$$\mathbb{Π}^*_{\mathbb{Π}} = \mathbb{Π}^* \left(r, a^0 \rightarrow a^*, \otimes(\mathbb{s}) = \varnothing\right)$$

with a^0 and a^* being the initial and goal steps of the plan corresponding to $\mathbb{s}_0$ such that $\mathit{eff}(a^0) = \mathit{pre}(\omega)$ and $\mathit{pre}(a^*) = \mathit{eff}(\omega)$. The iterator is all the possible resolutions of all flaws on any plan in the search space and the solution predicate is true when the plan has no more flaws.

Details about flaws, resolvers and the overall Partial Order Causal Links (POCL) algorithm will be presented in section 6.1.1.

This approach can usually give a partial plan if we set $t_{\mathbb{s}}$ too low for the algorithm to complete. This plan is not a solution but can eventually be used as an approximation for certain use cases (like intent recognition, see chapter 7).

4.5.3 Case based planning

Another plan oriented planning is called Case-Based Planning (CBP). This kind of planning relies on a library $\mathcal{C}$ of already complete plans and try to find the most appropriate one to repair. To repair a plan we use a derivative of the refinement function noted r_+ that will make either a classical refinement or repair a part of the plan.

tries

$$\mathbb{Π}_{\mathcal{C}} = \mathbb{Π}^* \left(r_+, \{\!|\min : \mathbb{π} \in \mathcal{C} \wedge \mathbb{π}(\mathit{pre}(\omega)) \vDash \mathit{eff}(\omega)|\!\}, \mathbb{s}(\mathit{pre}(\omega)) \neq \varnothing\right)$$

The planner selects a plan that fits the best efficiently with the initial and goal state of the problem. This plan is then repaired and validated iteratively. The problem with this approach is that it may be unable to find a valid plan or might need to populate and maintain a good plan library. For such case an auxiliary planner is used (preferably a diverse planner; that gives several solution).

4.5.4 Probabilistic planning

Probabilistic planning tries to deal with uncertainty by working on a policy instead of a plan. The initial problem holds probability laws that govern the execution of any actions. It is sometimes accompanied with a reward function instead of a deterministic goal. We use the set of states as search space with the policy as the iterator.

$$\mathbb{\Pi}_{\mathbb{P}} = \mathbb{\Pi}^* (\mathbb{pol}, \mathit{pre}(\omega), \mathbb{s} \vDash \mathit{eff}(\omega))$$

At each iteration a state is chosen from the frontier. The frontier is updated with the application of the most likely to succeed action given by the policy. The search stops when the frontier satisfies the goal.

4.5.5 Hierarchical planning

Hierarchical Task Networks (HTN) are a totally different kind of planning paradigm. Instead of a goal description, HTN uses a root task that needs to be decomposed. The task decomposition is an operation that replaces a task (action) by one of its methods $\mathbb{\Pi}$.

$$\mathbb{\Pi}_{\omega} = \mathbb{\Pi}^* ((\mathbb{\Pi}, \mathbb{s} \rightarrow \{\!\{\mathbb{\pi} \in \mathbb{\Pi}(\{\!\{a \in A_{\mathbb{s}} \wedge \mathbb{\Pi}(a) \neq \emptyset\}\!\})\}\!\}), \omega, \forall a \in A_{\mathbb{s}} : \mathbb{\Pi}(a) = \emptyset)$$

Un conclusion rapide: Nous avons detaillé ca et ca et c'est tout