6 Online and Flexible Planning Algorithms

In this chapter, we present planners and approaches to inverted planning and intent recognition. To do that we must first have a efficient online planning algorithm that can take into account observed plans or fluents and find the most likely plan to be pursued by an external agent. The planning process must be done in real time and take into account new observations to make new predictions. This requires the use of online planners. In such cases, the planning process works using distinct phases. In figure 6.1, we illustrate the components of the process. Only the planning part is meant to have real-time constraints on its execution the rest is usually a linear process and can be negligible in terms of execution times.

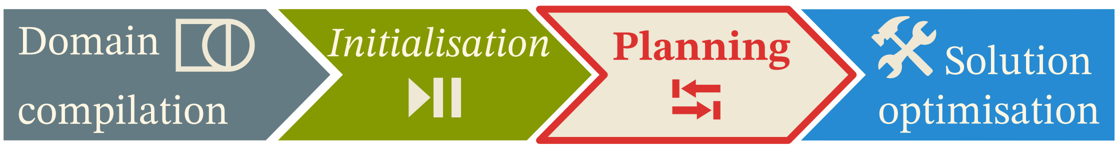


Figure 6.1: Planning phases for online planning

Classical planning can be used for such a work but lacks flexibility when needing to replan at high frequency. The planner must be either able to reuse previously found plans or be able to compute quickly plans that are good approximation of the intended goal. We could use probabilistic planning, especially Partially Observable Markovian Decision Process (POMDP) to directly encode the intent recognition problem but that approach has been explored in great detail already, including numerous Bayesian network approaches. Further discussions of inverted planning and intent recognition can be found **LATER**.

It was decided to explore more expressive and flexible approaches to use the semantics of the planning domain to attempt to guide the search to a more logical plan. This approach uses either repair heuristics or explanation to provide fast predictions of intended goals.

6.1 Existing Algorithms

In order to make a planner capable of repairing plans, the most fitting paradigm is PSP as described **BEFORE**. Using the plan space for search allows to modify the refinement process into repairing existing plans.

The second approach using explanations is hierarchical. The planner will use a HTN planning domain that contains composite actions (or tasks) that have several methods (as plans) to realize them.

First, PSP will be presented in more details regarding its classical formulation and definition.

6.1.1 Plan Space Planning

All PSP algorithms work in a similar way: their search space is the set of all plans and its iteration operation is plan refinement. This means that every PSP planner searches for *flaws* in the current plan and then computes a set of *resolvers* that potentially fix each of them. The algorithm usually starts with an empty plan only having the initial and goal step and recursively refine the plan until all flaws have been solved.

In general, PSP is faster than naive classical planning. However, with the advent of efficient state based heuristics used in Fast Forward (FF) (Hoffmann 2001) and LAMA (Richter and Westphal 2010), plan space planning has been left behind regarding raw performance. While PSP delays commitment and therefore can make very efficient choices that can be faster than classical planning, most formulation of PSP problems leads to significant increase in complexity (Tan and Gruninger 2014). The backtracking in PSP algorithms along with heavy data structures such as plans to modify at each iteration makes the approach slower by design without an excellent heuristic.

Works on PSP didn't stop at that point (Nguyen and Kambhampati 2001) since it has unique advantages over classical planning. Indeed, by using backward chaining, PSP algorithms ~~have~~ are sound and complete and therefore guarantee to find a solution if it exists (Sjöberg and Nissar 2015).

As PSP finds partially ordered plans, it is also by nature more flexible. Indeed, multiple totally ordered plans are contained within a partially ordered one, they are called linearizations. So, when wanting to have several plans with low diversity, PSP is the way to go.

6.1.1.1 Definitions

In **BEFORE** we have formalized how PSP works in the general planning formalism. However, since this formalism is being introduced in the present document, it isn't used by the rest of the community. This means that we still need to define the classical Partially Ordered with Causal Link (POCL algorithm. In order to define this algorithm, we need to explain the notions of flaws and resolvers.)

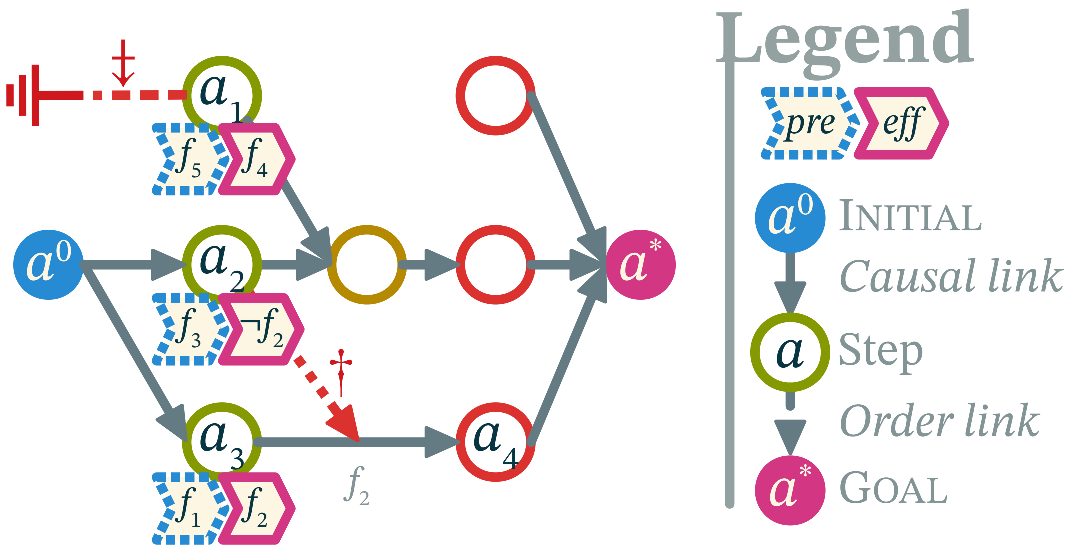
Definition 35 (Flaws)**.** Flaws are constraints violations within a plan. The set of flaws in a plan π is noted $\otimes_{\pi}$. There are different kinds of flaws in classical PSP and additional ones can be defined depending on the application.

Classical flaws often have a few common features. They are often **constructive** since they require an *addition* of causal links and steps in a plan to be fixed. They have a *proper fluent* f that is the cause of the violation in the plan the flaw is representing

TO

and a *needer* a_n that is the action requiring the proper fluent be fulfilled. In classical PSP flaws are either:

- **Subgoals**, also called *open condition* that are yet to be supported by a *provider* a_p. We note subgoals $\otimes^{\ddagger}_{a_n}(f)$.
- **Threats** are caused by steps that can break a causal link with their effects. They are called *breakers* of the threatened link. A step a_b threatens a causal link $l_t = a_p \xrightarrow{f} a_n$ if and only if $(eff(a_b) \not\models f) \wedge (a_b \not\succ a_p \wedge a_n \not\succ a_b)$. Said otherwise, the breaker can potentially cancel an effect of a providing step a_p, before it gets used by its needer a_n. We note threats $\otimes^{\dagger}_{a_n}(f, a_b)$.



il manque la légende pour subgoal et threat

Figure 6.2: Example of partial plan having flaws

Example 43. In figure 6.2 we present a partially ordered plan with two typical flaws. The first is a subgoal missing from the plan to fulfill the f_5 precondition of a_1.

The second flaw in the figure 6.2 is the threat between a_2 and a causal link outgoing from a_3. This happens because nothing prevents a_2 to be executed after a_3 and negate the fluent f_2 needed by the next step.

These flaws need to be fixed in order for the plan to be valid. In POCL it is done by finding their resolvers.

Definition 36 (Resolvers)**.** A resolver is a plan refinement that attempts to solve a flaw $\otimes_{a_n}$. Since classical flaws are constructive, the classical resolvers are called *positive*. They are defined as follows:

- *For subgoals*, the resolvers are a potential causal link containing the proper fluent f of a given subgoal in their causes while taking the needer step a_n as their target and a **provider** step a_p as their source. They are noted $\odot^{+}_{a_p}(\otimes^{\ddagger}_{a_n}(f)) = a_p \xrightarrow{f} a_n$.

- *For threats*, we usually consider only two resolvers: **demotion** ($a_b \succ a_p$) and **promotion** ($a_n \succ a_b$) of the breaker relative to the threatened link. We call the added causeless causal link a **guarding** link. The resolvers for threats are noted $\odot^{\pm}_{\succ}(\otimes^{\dagger}_{a_n}(f, a_b)) = a_p \rightarrow a_b$ for promotion and $\odot^{\pm}_{\prec}(\otimes^{\dagger}_{a_n}(f, a_b)) = a_b \rightarrow a_n$ for demotion.

It is possible to introduce extra resolvers to fix custom flaws. In such a case we call positive resolvers, those which add causal links and steps to the plan and negative those that removes causal links and steps. It is preferable to engineer flaws and resolver not to mix positive and negative aspect at once because of the complicated side effects that might result from it.

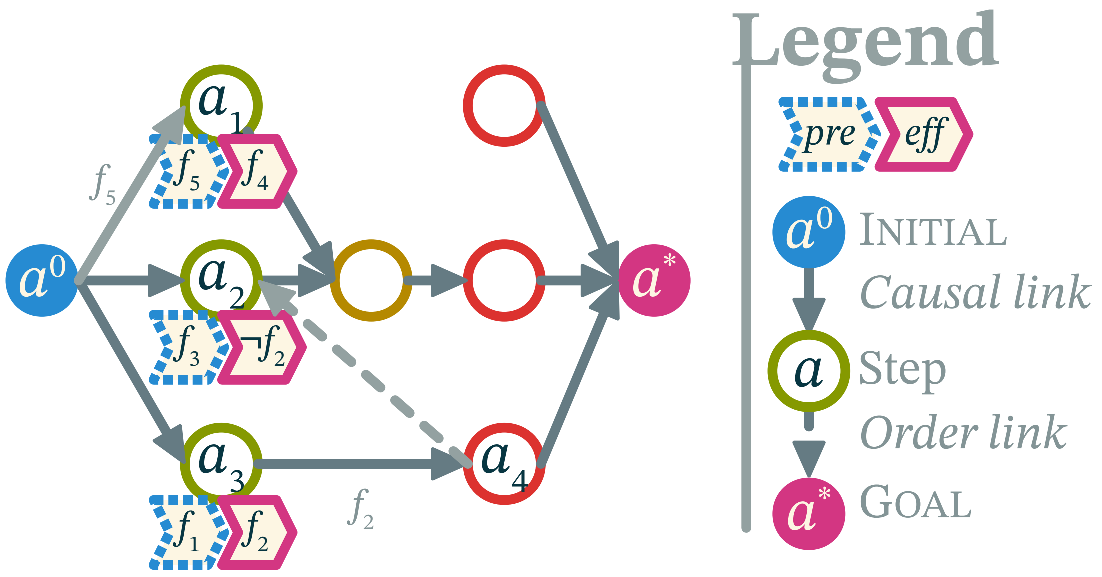


Figure 6.3: Example of resolvers that fixes the previously illustrated flaws

Example 44. From our previous example, we present the complete plan in figure 6.3. The subgoal needs to be fixed by inserting another causal link to provide the missing fluent and inserting any necessary steps to do so. In that case the initial state happens to provide the necessary fluent so we simply add a causal link for it.

For the threat, the solution is to either promote or demote a_2 so that it doesn't interfere with the causal link between a_3 and a_4. We chose here to demote a_2 so it requires a_4 to be executed before it.

The application of a resolver does not necessarily mean progress. It can have consequences that may require reverting its application in order to respect the backtracking of the POCL algorithm.

An agenda is a flaw container used for the flaw selection of POCL.

Definition 37 (Side effects)**.** Flaws that are caused by the application of a resolver are called *related flaws*. They are inserted into the *agenda* with each application of a resolver:

- *Related subgoals* are all the new open conditions inserted by new steps.

- *Related threats* are the causal links threatened by the insertion of a new step or the deletion of a guarding link.

Flaws can also become irrelevant when a resolver is applied. It is always the case for the targeted flaw, but this can also affect other flaws. Those *invalidated flaws* are removed from the agenda upon detection:

- *Invalidated subgoals* are subgoals satisfied by the new causal links or the removal of their needer.
- *Invalidated threats* happen when the breaker no longer threatens the causal link because the order guards the threatened causal link or either of them have been removed.

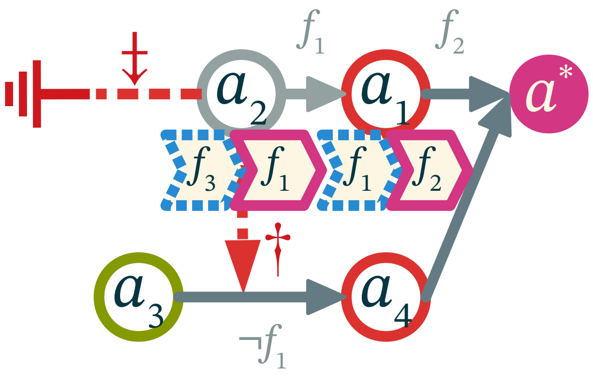


Figure 6.4: Example of the side effects of the application of a resolver

Example 45. In our example, by adding the step a_2 to fix an unsupported subgoal needed by a_1, we introduced another subgoal to support the new step that also threatens the causal link between a_3 and a_4.

6.1.1.2 Classical POCL Algorithm

In algorithm 5 we present a generic version of POCL inspired by Ghallab *et al.* (2004, sec. 5.4.2).

il manque la figure ???

For our version of POCL we follow a refinement procedure that works in several generic steps. In figure **??** we detail the resolution of a subgoal as done in the algorithm 5.

The first is the search for resolvers. It is often done in two separate steps: first, select the candidates and then check each of them for validity. This is done using the polymorphic function `solve` at line 5.

factory

In the case of subgoals, variable unification is performed to ensure the compatibility of the resolvers. Since this step is time-consuming, the operator is instantiated accordingly at this step to factories the computational effort. Composite operators have also all their methods instantiated at this step if they are selected as a candidate.

Then a resolver is picked non-deterministically for applications (this can be heuristically driven). At line 7 the resolver is effectively applied to the current plan. All side effects and invalidations are handled during the update of the agenda at line 8. If a

Algorithm 5 POCL Algorithm

```
1: function POCL(Agenda $\mathcal{A}$, Action $\omega$)
2:     if $\mathcal{A} = \emptyset$ then                          ▷ Populated agenda needs to be provided
3:         return Success                                         ▷ Stops all recursion
4:     Flaw $\otimes \leftarrow \{\!\!\{\mathcal{A}\}\!\!\}$                        ▷ Heuristically chosen flaw
5:     Resolvers $\bigodot \leftarrow$ SOLVE($\otimes$, $\{\!\!\{\Pi(\omega)\}\!\!\}$)   ▷ The root operator has only one method for PSP
6:     for all $\odot \in \bigodot$ do                              ▷ Non-deterministic choice operator
7:         APPLY($\odot$, $\pi$)                                   ▷ Apply resolver to partial plan
8:         Agenda $\mathcal{A}' \leftarrow$ UPDATE($\mathcal{A}$)
9:         if POCL($\mathcal{A}'$, $\omega$) = Success then             ▷ Refining recursively
10:            return Success
11:        REVERT($\mathcal{A}$, $\pi$)                              ▷ Failure, undo resolver application
12:    $\mathcal{A} \leftarrow \mathcal{A} \cup \{\otimes\}$                         ▷ Flaw was not resolved
13:    return Failure                                            ▷ Revert to last non-deterministic choice
```

problem occurs, line 11 backtracks and tries other resolvers. If no resolver fits the flaw, the algorithm backtracks to previous resolver choices to explore all the possible plans and ensure completeness.

6.1.1.3 Existing PSP Planners

Related works already tried to explore new ideas to make PSP an attractive alternative to regular state-based planners like the appropriately named "Reviving partial order planning" (Nguyen and Kambhampati 2001) and VHPOP (Younes and Simmons 2003). More recent efforts (Coles *et al.* 2011; Sapena *et al.* 2014) are trying to adapt the powerful heuristics from state-based planning to PSP's approach. An interesting approach of these last efforts is found in (Shekhar and Khemani 2016) with meta-heuristics based on offline training on the domain. Yet, we clearly note that only a few papers lay the emphasis upon plan quality using PSP (Ambite and Knoblock 1997; Say *et al.* 2016).

6.1.2 Plan Repair & Reuse

In online planning, the plan is computed frequently from a changing initial state. This means that the previous plan is very often available. In order to take advantage of the effort invested in previous plans, it is tempting to simply reuse the existing plan instead of replanning from scratch. Most work on the field focus on monitoring execution and find ways to make resilient plans.

In such a case, a lot of the planning model can be subject to uncertainty. Indeed, the execution of an action can fail because an external event changed a precondition required to do it or the model itself can be inaccurate.

The idea of reusing plan emerged early on (Nebel and Koehler 1995) but with a caveat: often repairing needed more effort than replanning. So plan repair became more of a gamble and needed incentives to reuse an existing plan given an application. For example, such process is useful for multi-agent planning where a significant change of plan is expansive among agents (Ephrati and Rosenschein 1993; Alami *et al.* 1995; Sugawara 1995; Borrajo 2013; Luis and Borrajo 2014). This motivation for plan repair comes from plan merging and is needed in cooperative environments.

The question of the efficiency of replanning vs. repairing has been since studied extensively under several aspects (Van Der Krogt and De Weerdt 2005; Fox *et al.* 2006). The emergence of diverse planning and requirement on plan stability of execution monitoring gave new research on the subject. At this point, most of the literature focuses on a case-based planning, where a plan library is already provided and the planner must select a plan and repair it to fit a given case (Gerevini *et al.* 2013; Borrajo *et al.* 2015).

A recent work of Zhuo and Kambhampati (2017) gives an interesting approach to the problem by questioning the domain. Indeed, the need to re-plan can be an opportunity to revised issues in the current model and to improve it by adding newly found ~~a~~ solution to complete the given planning model.

In our case, we focus on PSP and how to make plans repair efficiently using that technique. Classical PSP algorithms don't take as an input an existing plan but can be enhanced to fit plan to repair, as for instance in (Van Der Krogt and De Weerdt 2005). Usually, PSP algorithms take a problem as an input and use a loop or a recursive function to refine the plan into a solution. We can't solely use the refining recursive function to be able to use our existing partial plan. This causes multiples side effects if the input plan is suboptimal. This problem was already explored as of LGP-adapt (Borrajo 2013) that explains how reusing a partial plan often implies replanning parts of the plan.

6.1.3 HTN

> "HTN planners differ from classical planners in what they plan for and how they plan for it. In an HTN planner, the objective is not to achieve a set of goals but instead to perform some set of tasks."

Ghallab *et al.* (2004)

Planning using HTN gives a completely different approach to the problem and its formulation. In this formalism, actions are composite tasks and there is no goal other than to complete the root task. One can find similarities with our general planning formalism and it isn't a coincidence. Indeed, HTN is more general than planning and therefore one need to be able to allow for this level of expressivity.

This expressivity comes at a cost. HTN problems are on a complexity category that is significantly harder than regular STRIPS planning:

From Bercher and Höller (2018) HTN Tutorial at ICAPS 2018

Table 6.1: HTN Expressivity associated with their respective complexity classes {#tbl:htn}

Restrictions	*Complexity*
Classical	PSPACE
Task Insertion	NEXPTIME
Totally Ordered	EXPTIME
Regular	PSPACE

Restrictions	*Complexity*
Acyclic	NEXPTIME
Tail-recursive	EXPSPACE

HTN is often combined with classical approaches since it allows for a more natural expression of domains making expert knowledge easier to encode. These kinds of planners are named **decompositional planners** when no initial plan is provided (Fox 1997). Most of the time the integration of HTN simply consists in calling another algorithm when introducing a composite operator during the planning process. The DUET planner by Gerevini *et al.* (2008) does so by calling an instance of a HTN planner based on the task insertion called SHOP2 (Nau *et al.* 2003) to decompose composite actions. Some planners take the integration further by making the decomposition of composite actions into a special step in their refinement process. Such works include the discourse generation oriented DPOCL (Young and Moore 1994) and the work of Kambhampati *et al.* (1998) generalizing the practice for decompositional planners.

In our case, we chose a class of hierarchical planners based on Plan-Space Planning (PSP) algorithms (Bechon *et al.* 2014; Dvorak *et al.* 2014; Bercher *et al.* 2014) as a reference approach. The main difference here is that the decomposition is integrated into the classical POCL algorithm by only adding new types of flaws. This allows keeping all the flexibility and properties of POCL while adding the expressivity and abstraction capabilities of HTN.

6.2 LOLLIPOP

That first planner is a prototype destined to test the feasibility of a plan to repair approach using PSP. This is done through the addition of new flaw types in a classical POCL algorithm.

Another issue is caused by the need to get any existing partial plan as an input. This plan can contain any problems and inconsistency since there are trivial ways to ensure consistence between the plan given and the new initial state or goal.

pas clair: s'il y a des manières triviales d'assurer la consistence, pourquoi le plan en entrée peut contenir des inconsistences ?

6.2.1 Operator Graph

In order to make POCL faster we experimented a means to build a dependency graph for operators in the domain, computed at domain compilation time. The idea is to extract a partial plan from this graph by pruning it based on the provided goal and use that plan to quick start the planning process. First we need to define the operator graph.

Definition 38 (Operator Graph)**.** An operator graph g_O of a set of operators O is a labeled directed graph that binds two operators with the causal link $o_1 \xrightarrow{f} o_2$ if and only if there exists at least one fluent so that $(f \in \mathit{eff}(o_1)) \land f \models \mathit{pre}(o_2)$.

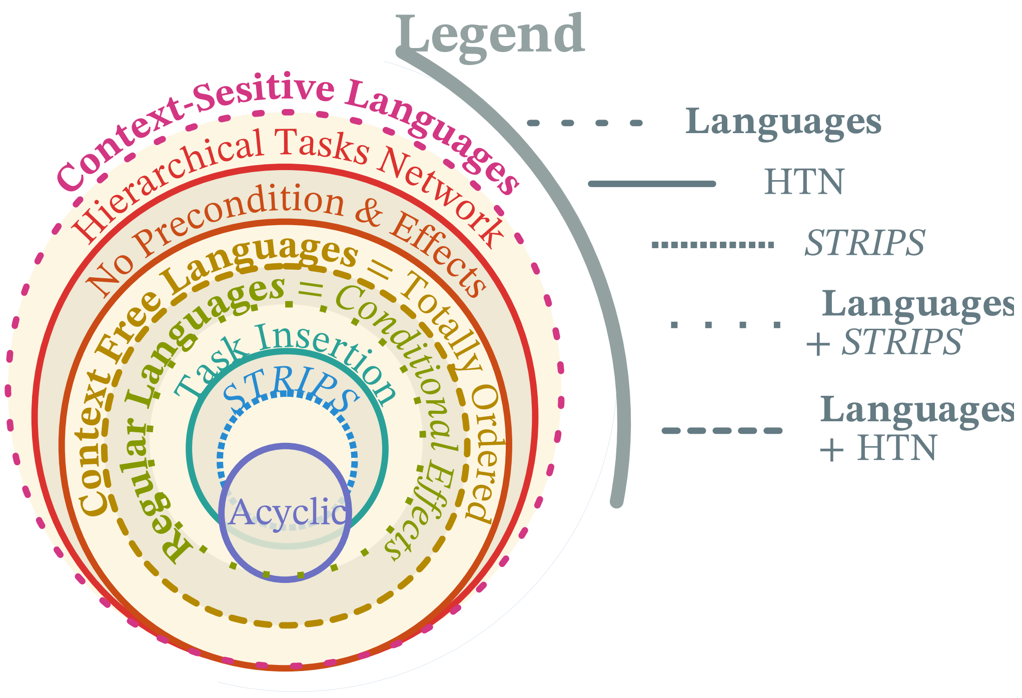

Legend
Languages
HTN
STRIPS
Languages + STRIPS
Languages + HTN
Context-Sesitive Languages
Hierarchical Tasks Network
No Precondition & Effects
Context Free Languages = Totally Ordered
Regular Languages = Conditional Effects
Task Insertion
STRIPS
Acyclic

Figure 6.5: Venn diagram of the expressivity class of HTN paradigms.

This definition was inspired by the notion of domain causal graph as explained in (Göbelbecker *et al.* 2010) and originally used as a heuristic in (Helmert *et al.* 2011). Causal graphs have fluents as their nodes and operators as their edges. Operator graphs are the opposite: an *operator dependency graph* for a set of actions. A similar structure was used in (Peot and Smith 1994) that builds the operator dependency graph of goals and uses precondition nodes instead of labels. We call *co-dependent* operators that form a cycle. If the cycle is made of only one operator (self-loop), then it is called *auto-dependent*.

While building this operator graph, we need a **providing map** that indicates, for each fluent, the list of operators that can provide it. This is a simpler version of the causal graphs that is reduced to an associative table easier to maintain. The list of providers can be sorted to drive resolver selection (as detailed in section ??). A **needing map** is also built but is only used for operator graph generation. We note $g_{\mathcal{D}}$ the operator graph built with the set of operators in the domain $\mathcal{D}$.

Example 46. In the figure 6.6, we illustrate the application of this mechanism on our example from figure ??. Continuous lines correspond to the *domain operator graph* computed during domain compilation time.

The generation of the operator graph is detailed in algorithm 6. It explores the operator space and builds a providing and a needing map that gives the provided and needed fluents for each operator. Once done it iterates on every precondition and searches for a satisfying cause to add the causal links to the operator graph.

To apply the notion of operator graphs to planning problems, we just need to add the initial and goal steps to the operator graph. In figure 6.6, we depict this insertion with

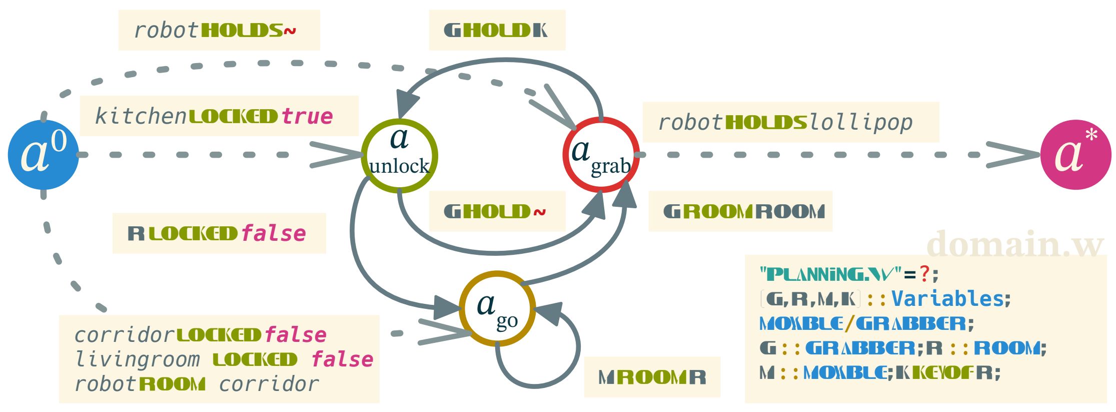


Figure 6.6: Diagram of the operator graph of example domain. Full arrows represent the domain operator graph and dotted arrows the dependencies added to inject the initial and goal steps.

Algorithm 6 Operator graph generation and update algorithm

```
function ADDVERTEX(Action $a$)
    CACHE($a$)                                  ▷ Update of the providing and needing map
    if binding then                             ▷ boolean that indicates if the binding was requested
        BIND($a$)
function CACHE(Action $a$)
    for all $f \in eff(a)$ do                   ▷ Adds $a$ to the list of providers of $f$
        ADD($A_p, f, a$)
    ...                                         ▷ Same operation with needing and preconditions
function BIND(Action $a$)
    for all $f \in pre(a)$ do
        if $f \in A_p$ then
            for all $\pi \in$ GET($A_p$, $f$) do
                Link $l \leftarrow$ GETEDGE($\pi$, $a$)      ▷ Create the link if needed
                ADDCAUSE($l$, $f$)                ▷ Add the fluent as a cause
    ...                                         ▷ Same operation with needing and effects
```

our previous example using dotted lines. However, since operator graphs may have cycles, they can't be used directly as input to POCL algorithms to ease the initial back chaining. Moreover, the process of refining an operator graph into a usable one could be more computationally expensive than POCL itself.

In order to give a head start to the LOLLIPOP algorithm, we propose to build operator graphs differently with the algorithm detailed in algorithm 7. A similar notion was already presented as "basic plans" in (Sebastia *et al.* 2000). These "basic" partial plans use a more complete but slower solution for the generation that ensures that each selected steps are *necessary* for the solution. In our case, we built a simpler solution that can solve some basic planning problems but that also makes early assumptions (since our algorithm can handle them). It does a simple and fast backward construction of a partial plan driven by the providing map. Therefore, it can be tweaked with the powerful heuristics of state search planning.

plus détailler l'algo 7

Algorithm 7 Safe operator graph generation algorithm

```
function SAFE(Action $\omega$)
    Stack<Action> $open \leftarrow [a^*]$
    Stack<Action> $closed \leftarrow \varnothing$
    while $open \neq \varnothing$ do
        Action $a \leftarrow$ POCL($open$)                          ▷ Remove $a$ from $open$
        PUSH($closed, a$)
        for all $f \in pre(a)$ do
            Actions $A_p \leftarrow$ GETPROVIDING($\Pi, f$)         ▷ Sorted by usefulness
            if $A_p = \varnothing$ then                            ▷ (see section ??)
                $S \leftarrow S \setminus \{\Pi\}$
                continue
            Action $a' \leftarrow$ GETFIRST($\Pi$)
            if $a' \in closed$ then
                continue
            if $a' \notin S$ then
                PUSH($open, a'$)
            $S \leftarrow S \cup \{a'\}$
            Link $l \leftarrow$ GETEDGE($a', a$)                     ▷ Create the link if needed
            ADDCAUSE($l, f$)                                  ▷ Add the fluent as a cause
```

This algorithm is useful since it is specifically used on goals. The result is a valid partial plan that can be used as input to POCL algorithms.

6.2.2 Negative Refinements

The classical POCL algorithm works upon a principle of positive plan refinements. The two standard flaws (subgoals and threats) are fixed by *adding* steps, causal links, or variable binding constraints to the partial plan. Online planning needs to be able to *remove* parts of the plan that are not necessary for the solution. Since we assume that the input partial plan is quite complete, we need to define new flaws to optimize and fix this plan. These flaws are called *negative* as their resolvers apply subtractive refinements on partial plans.

Definition 39 (Alternative). An alternative is a negative flaw that occurs when there is a better provider choice for a given link. An alternative to a causal link $a_p \xrightarrow{f} a_n$ is a provider a_b that has a better *utility value* than a_p.

The **utility value** of an operator is a measure of usefulness being the base of our ranking mechanism detailed in section ??. It uses the incoming and outgoing degrees of the operator in the domain operator graph to measure its usefulness.

Finding an alternative to an operator is computationally expensive. It requires searching a better provider for every fluent needed by a step. To simplify that search, we select only the best provider for a given fluent and check if the one used is the same. If not, we add the alternative as a flaw. This search is done only on updated steps for online planning. Indeed, the safe operator graph mechanism is guaranteed to only choose the best provider (algorithm 7 at line 12). Furthermore, subgoals won't introduce new fixable alternatives as they are guaranteed to select the best possible provider.

Definition 40 (Orphan). An orphan is a negative flaw that occurs when a step in the partial plan (other than the initial or goal step) is not participating in the plan. Formally a_o is an orphan if and only if $a_o \neq a^0 \wedge a_o \neq a^* \wedge (|\chi_{\bullet\!\!\prec}(a_o)| = 0) \vee \{\!\!\{= (\varnothing) : \chi_{\bullet\!\!\prec}(a_o)\}\!\!\}$.

With $\chi_{\bullet\!\!\prec}(a_o)$ being the set of *outgoing causal links* of a_o in π. This last condition checks for *dangling orphans* that are linked to the goal with only bare causal links (introduced by threat resolution).

The solution to an alternative is a negative refinement that simply removes the targeted causal link. This causes a new subgoal as a side effect, which will focus on its resolver by its rank (explained in section ??) and then pick the first provider (the most useful one). The resolver for orphans is the negative refinement that is meant to remove a step and its incoming causal link while tagging its providers as potential orphans.

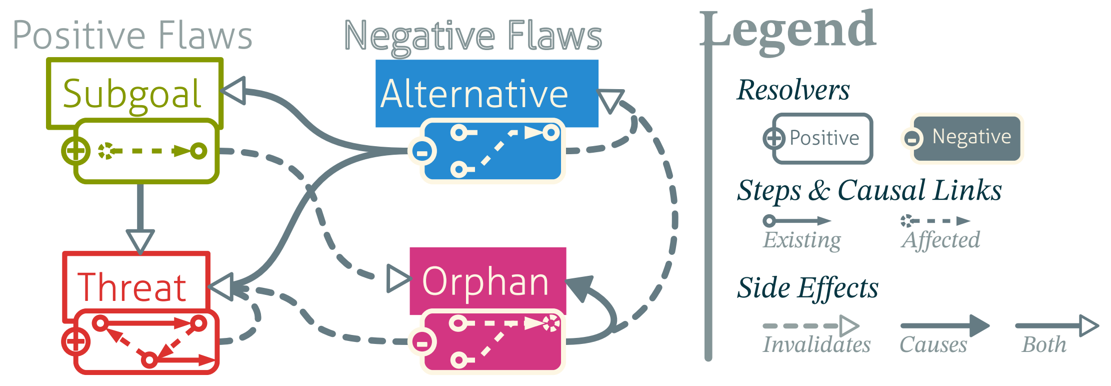


Figure 6.7: Schema representing flaws with their signs, resolvers and side effects relative to each other

The side effects mechanism also needs an upgrade since the new kinds of flaws can interfere with one another. This is why we extend the side effect definition (cf. definition ??) with a notion of sign.

Definition 41 (Signed Side Effects)**.** A signed side effect is either a regular *causal side effect* or an *invalidating side effect*. The sign of a side effect indicates if the related flaw needs to be added or removed from the agenda.

The figure 6.7 exposes the extended notion of signed resolvers and side effects. When treating positive resolvers, nothing needs to change from the classical method. When dealing with negative resolvers, we need to search for extra subgoals and threats. Deletion of causal links and steps can cause orphan flaws that need to be identified for removal.

In the method described in (Peot and Smith 1993), a **invalidating side effect** is explained under the name of *DEnd* strategy. In classical POCL, it has been noticed that threats can disappear in some cases if subgoals or other threats were applied before them. For our mechanisms, we decide to gather under this notion every side effect that removes the need to consider a flaw. For example, orphans can be invalidated if a subgoal selects the considered step. Alternatives can remove the need to compute further subgoal of an orphan step as orphans simply remove the need to fix any flaws that concern the selected step.

These interactions between flaws are decisive for the validity and efficiency of the whole model, that is why we aim to drive flaw selection in a rigorous manner.

6.2.3 Usefulness Heuristic

Resolvers and flaws selection are the keys to improving performances. Choosing a good resolver helps to reduce the branching factor that accounts for most of the time spent on running POCL algorithms (Kambhampati 1994). Flaw selection is also important for efficiency, especially when considering negative flaws which can conflict with other flaws.

Conflicts between flaws occur when two flaws of opposite sign target the same element of the partial plan. This can happen, for example, if an orphan flaw needs to remove a step needed by a subgoal or when a threat resolver tries to add a promoting link against an alternative. The use of side effects will prevent most of these occurrences in the agenda but a base ordering will increase the general efficiency of the algorithm.

Based on the figure 6.7, we define a base ordering of flaws by type. This order takes into account the number of flaw types affected by causal side effects.

1. **Alternatives** will cut causal links that have a better provider. It is necessary to identify them early since they will add at least another subgoal to be fixed as a related flaw.
2. **Subgoals** are the flaws that cause most of the branching factor in POCL algorithms. This is why we need to make sure that all open conditions are fixed before proceeding on finer refinements.

3. **Orphans** remove unneeded branches of the plan. Yet, these branches can be found out to be necessary for the plan to meet a subgoal. Since a branch can contain many actions, it is preferable to leave the orphan in the plan until they are no longer needed. Also, threats involving orphans are invalidated if the orphan is resolved first.
4. **Threats** occur quite often in the computation. Searching and solving them is computationally expensive since the resolvers need to check if there are no paths that fix the flaw already. Many threats are generated without the need of resolver application (Peot and Smith 1993). That is why we rank all related subgoals and orphans before threats because they can add causal links or remove threatening actions that will fix the threat.

Resolvers need to be ordered as well, especially for the subgoal flaws. Ordering resolvers for a subgoal is the same operation as choosing a provider. Therefore, the problem becomes "how to rank operators?". Usually, each operator has an assigned cost in the domain, but more often than not, costs are hard to estimate manually. In our case we need an automated way to rank operators. The most relevant information on an operator is how useful it may be to other actions in the plan and how hard is it to realize.

Since this may be computationally expensive to compute while planning, the evaluation of the cost of an operator is done offline using the operator graph.

The first metric to compute this heuristic is the degree of the operator.

Definition 42 (Degree of an operator)**.** Degrees are a measurement of the usefulness of an operator. Such a notion is derived from the incoming and outgoing degrees of a node in the operator graph.

We note $|\chi_{\bullet\!\!\prec}(a)|$ being the *outgoing degree* of a in the directed graph formed by ℿ and $|\chi_{\succ\!\!\bullet}(a)|$ being the *incoming degree* of a in the directed graph formed by ℿ respectively the outgoing and incoming degrees of an operator in a plan ℿ. These represent the number of causal links that goes out or toward the operator. We call proper degree of an operator $|\mathit{eff}(a)|$ and $|\mathit{pre}(a)|$ the number of preconditions and effects that reflect its intrinsic usefulness.

There are several ways to use the degrees as indicators. The *utility value* increases with every outgoing degree, since this reflects a positive participation in the plan. It decreases with every negative degree since actions with higher incoming degrees are harder to satisfy. The utility value bounds are useful when selecting special operators. For example, a user-specified constraint could be laid upon an operator to ensure it is only selected as a last resort. This operator will be affected with the smallest utility value possible. More commonly, the highest value is used for initial and goal steps to ensure their selection.

In this section χ is the connectivity of the operator graph g_O.

Our ranking mechanism is based on scores noted ₹(a). A score is a tuple of metrics:

- ₹$_1(a) = |\chi_{\bullet\!\!\prec}(a)|$ is the positive degree of a in the domain operator graph. This will give a measurement of the predicted usefulness of the operator.

- $\text{₹}_2(a) = |\otimes_a^{\ddagger}|$ is the number of open conditions of a in the domain operator graph. This is symptomatic of action that can't be satisfied without a compliant initial step.
- $\text{₹}_3(a) = |pre(a)|$ is the proper negative degree of a. Having more preconditions will likely add subgoals.
- $\text{₹}_4(a) = \{\!\!\{\min_{+\infty}(n) : n \in \mathbb{N} \land (a \to a) \in \chi^n \land \chi^n \neq \chi^+\}\!\!\}$ is the size of the shortest cycle involving a in the operator graph or $+\infty$ if there is none. Having this value at 1 is usually symptomatic of a *toxic operator* (cf. definition 44). Having an operator behaving this way can lead to backtracking because of operator instantiation.

The computation of the cost of the operator is done by multiplying the score tuple with a weighted parameter tuple α given by the user. The cost is then:

$$¢(a) = -\sum_{i=1}^{4} \alpha_i \text{₹}_i(a)$$

In practice, α_1 is positive, and the rest is negative. It is also better to make sure that
go $-1 \leq \alpha_4 \leq 0$ so that the penalties goes down as the cycles get~~s~~ bigger.

This respects the criteria of having a bound for the *utility value* as it ensures that it remains positive with 0 as a minimum bound and $+\infty$ for a maximum. The initial and goal steps have their utility values set to the upper bound to ensure their selection over other steps.

Choosing to compute the resolver selection at operator level has some positive consequences on the performances. Indeed, this computation is much lighter than approaches with heuristics on plan space (Shekhar and Khemani 2016) as it reduces the overhead caused by real time computation of heuristics on complex data. In order to reduce this overhead more, the algorithm sorts the providing associative array to easily retrieve the best operator for each fluent. This means that the evaluation of the heuristic is done only once for each operator. This reduces the overhead and allows for faster results on smaller plans.

6.2.4 Algorithm

The LOLLIPOP algorithm uses the same refinement algorithm as described in algorithm 5. The differences reside in the changes made on the behavior of resolvers and side effects. In line 7 of algorithm 5, LOLLIPOP algorithm applies negative resolvers if the selected flaw is negative. In **line ??**, it searches for both signs of side effects. Another change resides in the initialization of the solving mechanism and the domain as detailed in algorithm 8. This algorithm contains several parts. First, the `domainInit` function corresponds to the code computed during the domain compilation time. It will prepare the rankings, the operator graph, and its caching mechanisms. It will also use strongly connected component detection algorithm to detect cycles. These cycles are used during the base score computation (line 11). We add a detection of illegal fluents and operators in our domain initialization (line 5). Illegal operators are either inconsistent or toxic.

Algorithm 8 LOLLIPOP initialization (preprocessing) mechanisms

```
function DOMAININIT(Operators $A$)
    operatorgraph $g_O$
    Score $S$
    for all Operator $a \in A$ do
        if ISILLEGAL($a$) then                                ▷ Remove toxic and useless fluents
            $A \leftarrow A \setminus \{a\}$                  ▷ If entirely toxic or useless
            continue
        ADDVERTEX($a, g_O$)                                   ▷ Add and bind all operators
        CACHE($p, a$)                                         ▷ Cache operator in providing map
    Cycles $C \leftarrow$ STRONGLYCONNECTEDCOMPONENT($g_O$)   ▷ Using DFS
    $S \leftarrow$ BASESCORES($A$, $\mathcal{D}^\Pi$)
    $i \leftarrow$ INAPPLICABLES($\mathcal{D}^\Pi$)
    $e \leftarrow$ EAGERS($\mathcal{D}^\Pi$)
function LOLLIPOPINIT(Problem $\mathcal{P}$)
    REALIZE($S, \mathcal{P}$)                                 ▷ Realize the scores
    CACHE($providing, I$)                                     ▷ Cache initial step in providing ...
    CACHE($providing, G$)                                     ▷ ... as well as goal step
    SORT($providing, S$)                                      ▷ Sort the providing map
    if $L = \emptyset$ then
        $\mathcal{P}^\Pi \leftarrow$ SAFE($\mathcal{P}$)      ▷ Computing the safe operator graph if the plan is empty
    POPULATE($a$, $\mathcal{P}$)                              ▷ populate agenda with first flaws
function POPULATE(Agenda $a$, Problem $\mathcal{P}$)
    for all Update $u \in U$ do                               ▷ Updates due to online planning
        Fluents $F \leftarrow eff(u.new) \setminus eff(u.old)$   ▷ Added effects
        for all Fluent $f \in F$ do
            for all Operator $o \in$ BETTER($providing, f, o$) do
                for all Link $l \in L^+(o)$ do
                    if $f \in l$ then
                        ADDALTERNATIVE($a, f, o, l_\leftarrow, \mathcal{P}$)   ▷ With $l_\leftarrow$ the target of $l$
        $F \leftarrow eff(u.old) \setminus eff(u.new)$        ▷ Removed effects
        for all Fluent $f \in F$ do
            for all Link $l \in L^+(u.new)$ do
                if ISLIAR($l$) then
                    $L \leftarrow L \setminus \{l\}$
                    ADDORPHANS($a, u, \mathcal{P}$)
        ...                                                   ▷ Same with removed preconditions and incomming liar links
    for all Operator $o \in S$ do
        ADDSUBGOALS($a, o, \mathcal{P}$)
        ADDTHREATS($a, o, \mathcal{P}$)
```

Definition 43 (Inconsistent operators)**.** An operator a is contradictory iff $\exists f \{f, \neg f\} \in eff(o) \vee \{f, \neg f\} \in pre(o)$.

Definition 44 (Toxic operators)**.** Toxic operators have effects that are already in their preconditions or empty effects. An operator o is toxic iff $pre(o) \cap eff(o) \neq \emptyset \vee eff(o) = \emptyset$.

Toxic actions can damage a plan as well as make the execution of POCL algorithm longer than necessary. This is fixed by removing the toxic fluents ($pre(a) \not\subseteq eff(a)$) and by updating the effects with $eff(a) = eff(a) \setminus pre(a)$. If the effects become empty, the operator is removed from the domain.

problème de police de caractère

The lollipopInit function is executed during the initialization of the solving algorithm. We start by realizing the scores, then we add the initial and goal steps in the providing map by caching them. Once the ranking mechanism is ready, we sort the providing map. With the ordered providing map, the algorithm runs the fast generation of the safe operator graph for the problem's goal.

The last part of this initialization (line 21) is the agenda population that is detailed in the populate function. During this step, we perform a search of alternatives based on the list of updated fluents. Online updates can make the plan outdated relative to the domain. This forms liar links :

Definition 45 (Liar links)**.** A liar link is a link that doesn't hold a fluent in the preconditions or effect of its source and target. We note:

$$a_i \xrightarrow{f} a_j | f \notin eff(a_i) \cap pre(a_j)$$

A liar link can be created by the removal of an effect or preconditions during online updates (with the causal link still remaining).

We call lies the fluents that are held by links without being in the connected operators. To resolve the problem, we remove all lies. We delete the link altogether if it doesn't bear any fluent as a result of this operation. This removal triggers the addition of orphan flaws as side effects.

While the list of updated operators is crucial for solving online planning problems, a complementary mechanism is used to ensure that LOLLIPOP is complete. User provided plans have their steps tagged. If the failure has backtracked to a user-provided step, then it is removed and replaced by subgoals that represent each of its participation in the plan. This mechanism loops until every user provided steps have been removed.

6.2.5 Theoretical and Empirical Results

As proven in (Penberthy *et al.* 1992), the classical POCL algorithm is *sound* and *complete*.

First, we define some new properties of partial plans. The following properties are taken from the original proof. We present them again for convenience.

Definition 46 (Full Support)**.** A partial plan π is fully supported if each of its steps $o \in S$ is fully supported. A step is fully supported if each of its preconditions $f \in pre(o)$ is supported. A precondition is fully supported if there exists a causal link l that provides it. We note:

$$\Downarrow \pi \equiv \begin{array}{l} \forall o \in S\ \forall f \in pre(o)\ \exists l \in L_\pi^-(o) : \\ (f \in l \land \nexists t \in S(l_\rightarrow \succ t \succ o \land \neg f \in eff(t))) \end{array}$$

with $L_\pi^-(o)$ being the incoming causal links of o in π and $l_\rightarrow$ being the source of the link.

Definition 47 (Partial Plan Validity)**.** A partial plan is a **valid solution** of a problem $\mathcal{P}$ if it is *fully supported* and *contains no cycles*. The validity of π regarding a problem $\mathcal{P}$ is noted $\pi \vDash (\mathcal{P} \equiv \Downarrow \pi \land (C(\pi) = \emptyset))$ with $C(\pi)$ being the set of cycles in π.

6.2.5.1 Proof of Soundness

In order to prove that this property applies to LOLLIPOP, we need to introduce some hypothesis:

- operators updated by online planning are known.
- user provided steps are known.
- user provided plans don't contain illegal artifacts. This includes toxic or inconsistent actions, lying links and cycles.

Based on the definition 47 we state that:

$$\left(\begin{array}{l} \forall pre \in pre(G) : \\ \Downarrow pre \land \begin{array}{l} \forall o \in L_\pi^-(G)_\rightarrow\ \forall pre' \in pre(o) : \\ (\Downarrow pre' \land C_o(\pi) = \emptyset) \end{array} \end{array} \right) \implies \pi \vDash \mathcal{P}$$

{#eq:recursivevalidity} where $L_\pi^-(G)_\rightarrow$ is the set of direct antecedents of G and $C_o(\pi)$ is the set of fluents containing o in π.

This means that π is a solution if all preconditions of G are satisfied. We can satisfy these preconditions using operators if and only if their preconditions are all satisfied and if there is no other operator that threatens their supporting links.

First, we need to prove that equation **??** holds on LOLLIPOP initialization. We use our hypothesis to rule out the case when the input plan is invalid. The algorithm 7 will only solve open conditions in the same way subgoals do it. Thus, safe operator graphs are valid input plans.

Since the soundness is proven for regular refinements and flaw selection, we need to consider the effects of the added mechanisms of LOLLIPOP. The newly introduced refinements are negative, they don't add new links:

$$\forall f \in \mathcal{F}(\pi)\ \forall r \in r(f) : C_\pi(f.n) = C_{f(\pi)}(f.n)$$

{#eq:nocycle} with $\mathcal{F}(\pi)$ being the set of flaws in π, $r(f)$ being the set of resolvers of f, $f.n$ being the needer of the flaw and $f(\pi)$ being the resulting partial plan after the application of the flaw. Said otherwise, an iteration of LOLLIPOP won't add cycles inside a partial plan.

The orphan flaw targets steps that have no path to the goal and so can't add new open conditions or threats. The alternative targets existing causal links. Removing a causal link in a plan breaks the full support of the target step. This is why an alternative will always insert a subgoal in the agenda corresponding to the target of the removed causal link. Invalidating side effects also doesn't affect the soundness of the algorithm since the removed flaws are already solved. This makes:

$$\forall f \in \mathcal{F}^-(\pi) : \Downarrow \pi \implies \Downarrow f(\pi)$$

{#eq:conssupport} with $\mathcal{F}^-(\pi)$ being the set of negative flaws in the plan π. This means that negative flaws don't compromise the full support of the plan.

Equation **??** lead to equation **??** being valid after the execution of LOLLIPOP. The algorithm is sound.

6.2.5.2 Proof of Completeness

The soundness proof shows that LOLLIPOP's refinements don't affect the support of plans in terms of validity. It was proven that POCL is complete. There are several cases to explore to transpose the property to LOLLIPOP:

Lemma (Conservation of Validity). *If the input plan is a valid solution, LOLLIPOP returns a valid solution.*

Proof. With equation **??** and the proof of soundness, the conservation of validity is already proven. □

Lemma (Reaching Validity with incomplete partial plans). *If the input plan is incomplete, LOLLIPOP returns a valid solution if it exists.*

Proof. Since POCL is complete and the equation **??** proves the conservation of support by LOLLIPOP, then the algorithm will return a valid solution if the provided plan is an incomplete plan and the problem is solvable. □

Lemma (Reaching Validity with empty partial plans). *If the input plan is empty and the problem is solvable, LOLLIPOP returns a valid solution.*

Proof. This is proven using **BEFORE** and POCL's completeness. However, we want to add a trivial case to the proof: $pre(G) = \emptyset$. In this case the line **??** of the algorithm 5 will return a valid plan.

□

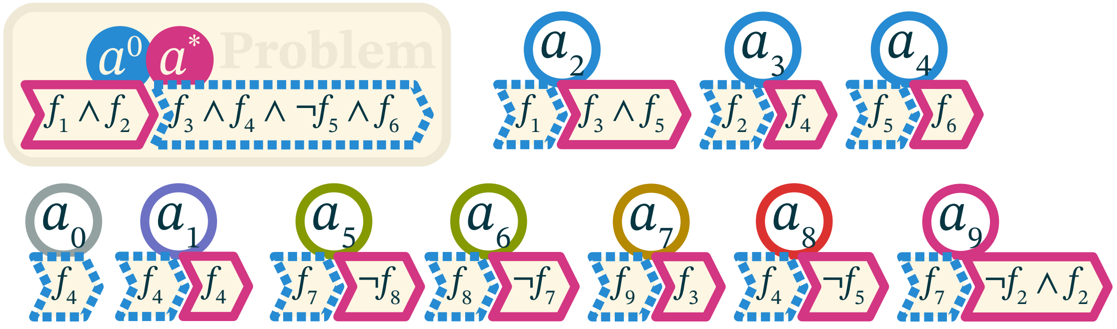
Lemma (Reaching Validity with a dead-end partial plan). *If the input plan is in a dead-end, LOLLIPOP returns a valid solution.*

Proof. Using input plans that can be in an undetermined state is not covered by the original proof. The problem lies in the existing steps in the input plan. Still, using our hypothesis we add a failure mechanism that makes LOLLIPOP complete. On failure, the needer of the last flaw is deleted if it wasn't added by LOLLIPOP. User defined steps are deleted until the input plan acts like an empty plan. Each deletion will cause corresponding subgoals to be added to the agenda. In this case, the backtracking is preserved and all possibilities are explored as in POCL. □

Since all cases are covered, this proves the property of completeness.

6.2.5.3 Experimental Results

The experimental results focused on the properties of LOLLIPOP for online planning. Since classical POCL is unable to perform online planning, we tested our algorithm considering the time taken for solving the problem for the first time. We profiled the algorithm on a benchmark problem containing each of the possible issues described earlier.



c'est la bonne Figure ? il n'y a pas de t, n, l … comme dans ta légende !

Figure 6.8: Domain used to compute the results. First line is the initial and goal steps along with the useful actions. Second line contains a threatening action t, two co-dependent actions n and l, a useless action u, a toxic action v, a dead-end action w and an inconsistent action x

In figure 6.8, we expose the planning domain used for the experiments. During the domain initialization, the actions u and v are eliminated from the domain since they serve no purpose in the solving process. The action x is stripped of its negative effect because it is inconsistent with the effect f_2.

As the solving starts, LOLLIPOP computes a safe operator graph (full lines in figure 6.9). As we can see, this partial plan is nearly complete already. When the main refining function starts it receives an agenda with only a few flaws remaining.

Then the main refinement function starts (time markers **1**). LOLLIPOP selects as resolver a causal link from a to satisfy the open condition of the goal step. Once the first threat between a and t is resolved the second threat is invalidated. On a second execution, the domain changes for online planning with 6 added to the initial step. This solving (time markers **2**) adds as flaw an alternative on the link from c to the goal step. A subgoal is added that links the initial and goal step for this fluent. An orphan

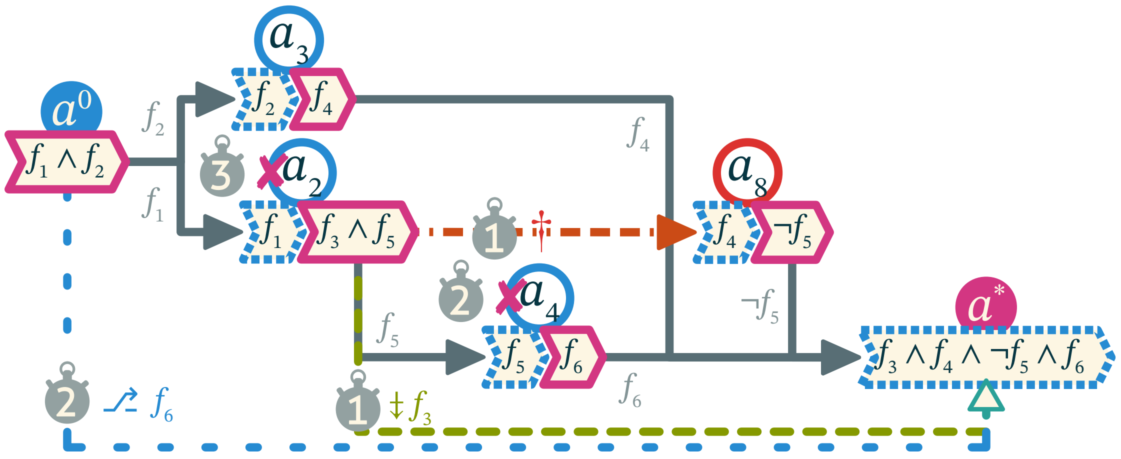


Figure 6.9: In full lines the initial safe operator graph. In thin, sparse and irregularly dotted lines respectively a subgoal, alternative and threat caused causal link.

flaw is also added that removes c from the plan. Another solving takes place as the goal step doesn't need 3 as a precondition (time markers **3**). This causes the link from a to be cut since it became a liar link. This adds a as an orphan that gets removed from the plan even if it was hanging by the bare link to t.

The measurements exposed in table **??** were made with an Intel® Core™ i7-4720HQ with a 2.60GHz clock. Only one core was used for the solving. The same experiment done only with the chronometer code gave a result of $70ns$ of errors. We can see an increase of performance in the online runs because of the way they are conducted by LOLLIPOP.

Table 6.2: Average times of 1.000 executions on the problem. The first column is for a simple run on the problem. Second and third columns are times to replan with one and two changes done to the domain for online planning. {#tbl:results}

Experiment	*Single*	*Online 1*	*Online 2*
Time (ms)	0.86937	0.38754	0.48123

6.3 HEART

6.3.1 Domain Compilation

In order to simplify the input of the domain, the causes of the causal links in the methods are optional. If omitted, the causes are inferred by unifying the preconditions and effects with the same mechanism as in the subgoal resolution in our POCL algorithm. Since we want to guarantee the validity of abstract plans, we need to ensure that user provided plans are solvable. We use the following formula to compute the final preconditions and effects of any composite action a: *pre*$(a) = \bigcup_{l \in L^+(I_m)}^{m \in methods(a)}$ *causes*(l) and *eff*$(a) = \bigcup_{l \in L^-(G_m)}^{m \in methods(a)}$ *causes*(l). An instance of the classical POCL algorithm is then run on the problem $\mathcal{P}_a = \langle \mathcal{D}, C_{\mathcal{P}}, a \rangle$ to ensure its coherence. The domain compilation fails if POCL cannot be completed. Since our decomposition hierarchy is acyclic ($a \notin A_a$, see definition 49) nested methods cannot contain their parent's action as a

$$a \rhd_\pi^- a' = \left\{ \phi^-(l) \xrightarrow{causes(l)} a' : l \in L_\pi^-(a) \right\}$$

It is the same with $a' \xrightarrow{causes(l)} \phi^+(l)$ and L^+ for $a \rhd^+ a'$. This supposes that the respective preconditions and effects of a and a' are equivalent. When not signed, the transposition is generalized: $a \rhd a' = a \rhd^- a' \cup a \rhd^+ a'$.

Example 47. $a \rhd^- a'$ gives all incoming links of a with the a replaced by a'.

Definition 49 (Proper Actions)**.** Proper actions are actions that are "contained" within an entity (either a domain, plan or action). We note this notion $A_a = A_a^{lv(a)}$ for an action a. It can be applied to various concepts:

- For a *domain* or a *problem*, $A_\mathcal{P} = A_\mathcal{D}$.
- For a *plan*, it is $A_\pi^0 = S_\pi$.
- For an *action*, it is $A_a^0 = \bigcup_{m \in methods(a)} S_m$. Recursively: $A_a^n = \bigcup_{b \in A_a^0} A_b^{n-1}$. For atomic actions, $A_a = \emptyset$.

Example 48. The proper actions of $make(drink)$ are the actions contained within its methods. The set of extended proper actions adds all proper actions of its single proper composite action $infuse(drink, water, cup)$.

Definition 50 (Abstraction Level)**.** This is a measure of the maximum amount of abstraction an entity can express, defined recursively by:

We use Iverson brackets here, see notations in table ??.

$$lv(x) = \left(\max_{a \in A_x}(lv(a)) + 1 \right) [A_x \neq \emptyset]$$

Example 49. The abstraction level of any atomic action is 0 while it is 2 for the composite action $make(drink)$. The example domain (in listing ??) has an abstraction level of 3.

The most straightforward way to handle abstraction in regular planners is illustrated by Duet (Gerevini *et al.* 2008) by managing hierarchical actions separately from a task insertion planner. We chose to add abstraction in POCL in a manner inspired by the work of Bechon *et al.* (2014) on a planner called HiPOP. The difference between the original HiPOP and our implementation of it is that we focus on the expressivity and the ways flaw selection can be exploited for partial resolution. Our version is lifted at runtime while the original is grounded for optimizations. All mechanisms we have implemented use POCL but with different management of flaws and resolvers. The original algorithm 5 is left untouched.

One of those changes is that resolver selection needs to be altered for subgoals. Indeed, as stated by the authors of HiPOP: the planner must ensure the selection of high-level operators in order to benefit from the hierarchical aspect of the domain. Otherwise, adding operators only increases the branching factor. Composite actions are not usually meant to stay in a finished plan and must be decomposed into atomic steps from one of their methods.

Definition 51 (Decomposition Flaws)**.** They occur when a partial plan contains a non-atomic step. This step is the needer a_n of the flaw. We note its decomposition $a_n \oplus$.

- *Resolvers:* A decomposition flaw is solved with a **decomposition resolver**. The resolver will replace the needer with one of its instantiated methods $m \in methods(a_n)$ in the plan π. This is done by using transposition such that: $a_n \oplus_\pi^m = \langle S_m \cup (S_\pi \setminus \{a\}), a_n \rhd^- I_m \cup a_n \rhd^+ G_m \cup (L_\pi \setminus L_\pi(a_n)\rangle$.
- *Side effects:* A decomposition flaw can be created by the insertion of a composite action in the plan by any resolver and invalidated by its removal:

$$\bigcup_{a_m \in S_m}^{f \in pre(a_m)} \pi' \ddagger_f a_m \bigcup_{a_b \in S_{\pi'}}^{l \in L_{\pi'}} a_b \boxtimes l \bigcup_{a_c \in S_m}^{lv(a_c) \neq 0} a_c \oplus$$

Example 50. When adding the step $make(tea)$ in the plan to solve the subgoal that needs tea being made, we also introduce a decomposition flaw that will need this composite step replaced by its method using a decomposition resolver. In order to decompose a composite action into a plan, all existing links are transposed to the initial and goal step of the selected method, while the composite action and its links are removed from the plan.

The main differences between HiPOP and HEART in our implementations are the functions of flaw selection and the handling of the results (one plan for HiPOP and a plan per cycle for HEART). In HiPOP, the flaw selection is made by prioritizing the decomposition flaws. Bechon *et al.* (2014) state that it makes the full resolution faster. However, it also loses opportunities to obtain abstract plans in the process.

6.3.3 Planning in cycle

The main focus of our work is toward obtaining **abstract plans** which are plans that are completed while still containing composite actions. In order to do that the flaw selection function will enforce cycles in the planning process.

Definition 52 (Cycle)**.** A cycle is a planning phase defined as a triplet $c = \langle lv(c), agenda, \pi_{lv(c)} \rangle$ where: $lv(c)$ is the maximum abstraction level allowed for flaw selection in the $agenda$ of remaining flaws in partial plan $\pi_{lv(c)}$. The resolvers of subgoals are therefore constrained by the following: $a_p \downarrow_f a_n : lv(a_p) \leq lv(c)$.

During a cycle all decomposition flaws are delayed. Once no more flaws other than decomposition flaws are present in the agenda, the current plan is saved and all remaining decomposition flaws are solved at once before the abstraction level is lowered for the next cycle: $lv(c') = lv(c) - 1$. Each cycle produces a more detailed abstract plan than the one before.

Abstract plans allow the planner to do an approximate form of anytime execution. At any given time the planner is able to return a fully supported plan. Before the first cycle, the plan returned is $\pi_{lv(a_0)}$.

Example 51. In our case using the method of intent recognition of Sohrabi *et al.* Sohrabi *et al.* (2016), we can already use $\pi_{lv(a_0)}$ to find a likely goal explaining an observation (a set of temporally ordered fluents). That can make an early assessment of the probability of each goal of the recognition problem.

For each cycle c, a new plan $\pi_{lv(c)}$ is created as a new method of the root operator a_0. These intermediary plans are not solutions of the problem, nor do they mean that the problem is solvable. In order to find a solution, the HEART planner needs to reach the final cycle c_0 with an abstraction level $lv(c_0) = 0$. However, these plans can be used to derive meaning from the potential solution of the current problem and give a good approximation of the final result before its completion.

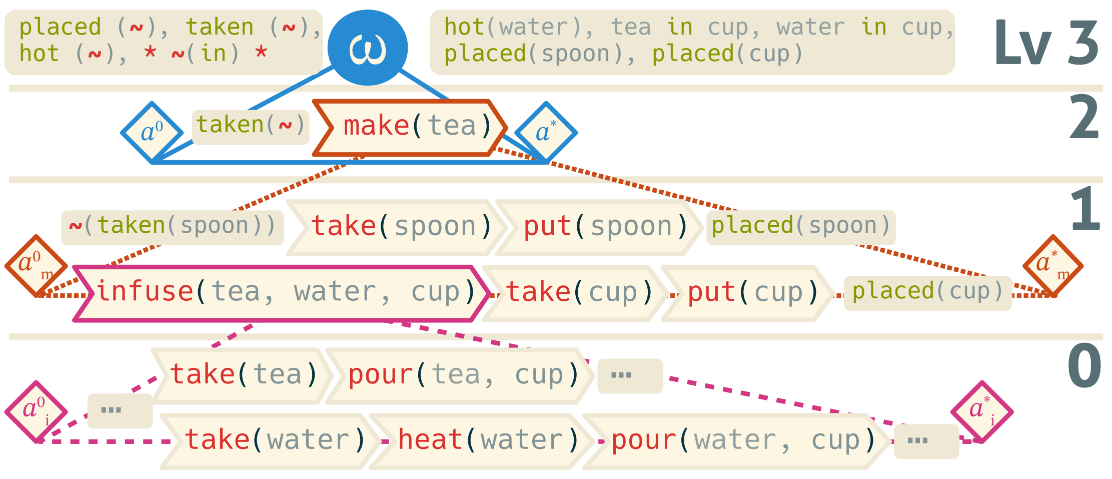


Figure 6.10: Illustration of how the cyclical approach is applied on the example domain. Atomic actions that are copied from a cycle to the next are omitted.

Example 52. In the figure 6.10, we illustrate the way our problem instance is progressively solved. Before the first cycle c_2, all we have is the root operator and its plan π_3. Then within the first cycle, we select the composite action $make(tea)$ instantiated from the operator $make(drink)$ along with its methods. All related flaws are fixed until all that is left in the agenda is the abstract flaws. We save the partial plan π_2 for this cycle and expand $make(tea)$ into a copy of the current plan π_1 for the next cycle. The solution of the problem will be stored in π_0 once found.

6.3.4 Properties of Abstract Planning

In this section, we prove several properties of our method and resulting plans: HEART is complete, sound and its abstract plans can always be decomposed into a valid solution.

The completeness and soundness of POCL has been proven in (Penberthy *et al.* 1992). An interesting property of POCL algorithms is that flaw selection strategies do not impact these properties. Since the only modification of the algorithm is the extension of the classical flaws with a decomposition flaw, all we need to explore, to update the proofs, is the impact of the new resolver. By definition, the resolvers of decomposition flaws will take into account all flaws introduced by its resolution into the refined plan. It can also revert its application properly.

Lemma (Decomposing preserves acyclicity). *The decomposition of a composite action with a valid method in an acyclic plan will result in an acyclic plan. Formally* $\forall a_s \in S_\pi : a_s \not\succ_\pi a_s \implies \forall a'_s \in S_{a \oplus^m_\pi} : a'_s \not\succ_{a \oplus^m_\pi} a'_s$.

Proof. When decomposing a composite action a with a method m in an existing plan π, we add all steps S_m in the refined plan. Both π and m are guaranteed to be cycle free by definition. We can note that $\forall a_s \in S_m : \left(\nexists a_t \in S_m : a_s \succ a_t \land \neg f \in \mathit{eff}(a_t)\right) \implies f \in \mathit{eff}(a)$. Said otherwise, if an action a_s can participate a fluent f to the goal step of the method m then it is necessarily present in the effects of a. Since higher level actions are preferred during the resolver selection, no actions in the methods are already used in the plan when the decomposition happens. This can be noted $\exists a \in \pi \implies S_m \uplus S_\pi$ meaning that in the graph formed both partial plans m and π cannot contain the same edges therefore their acyclicity is preserved when inserting one into the other.

□

Lemma (Solved decomposition flaws cannot reoccur). *The application of a decomposition resolver on a plan* π*, guarantees that* $a \notin S_{\pi'}$ *for any partial plan refined from* π *without reverting the application of the resolver.*

Proof. As stated in the definition of the methods (definition 30): $a \notin A_a$. This means that a cannot be introduced in the plan by its decomposition or the decomposition of its proper actions. Indeed, once a is expanded, the level of the following cycle $c_{lv(a)-1}$ prevents a to be selected by subgoal resolvers. It cannot either be contained in the methods of another action that are selected afterward because otherwise following definition 50 its level would be at least $lv(a) + 1$.

□

Lemma (Decomposing to abstraction level 0 guarantees solvability). *Finding a partial plan that contains only decomposition flaws with actions of abstraction level 1, guarantees a solution to the problem.*

Proof. Any method m of a composite action $a : lv(a) = 1$ is by definition a solution of the problem $\mathcal{P}_a = \langle \mathcal{D}, C_\mathcal{P}, a \rangle$. By definition, $a \notin A_a$, and $a \notin A_{a \oplus^m_\pi}$ (meaning that a cannot reoccur after being decomposed). It is also given by definition that the instantiation of the action and its methods are coherent regarding variable constraints (everything is instantiated before selection by the resolvers). Since the plan π only has decomposition flaws and all flaws within m are guaranteed to be solvable, and both are guaranteed to be acyclical by the application of any decomposition $a \oplus^m_\pi$, the plan is solvable.

□

Lemma (Abstract plans guarantee solvability). *Finding a partial plan* π *that contains only decomposition flaws, guarantees a solution to the problem.*

Proof. Recursively, if we apply the previous proof on higher level plans we note that decomposing at level 2 guarantees a solution since the method of the composite actions are guaranteed to be solvable.

□

From these proofs, we can derive the property of soundness (from the guarantee that the composite action provides its effects from any methods) and completeness (since if a composite action cannot be used, the planner defaults to using any action of the domain).

6.3.5 Computational Profile

In order to assess its capabilities, HEART was evaluated on two criteria: quality and complexity. All tests were executed on an Intel® Core™ i7-7700HQ CPU clocked at 2.80GHz. The Java process used only one core and was not limited by time or memory (32 GB that wasn't entirely used up) . Each experiment was repeated between 700 and 10000 times to ensure that variations in speed were not impacting the results.

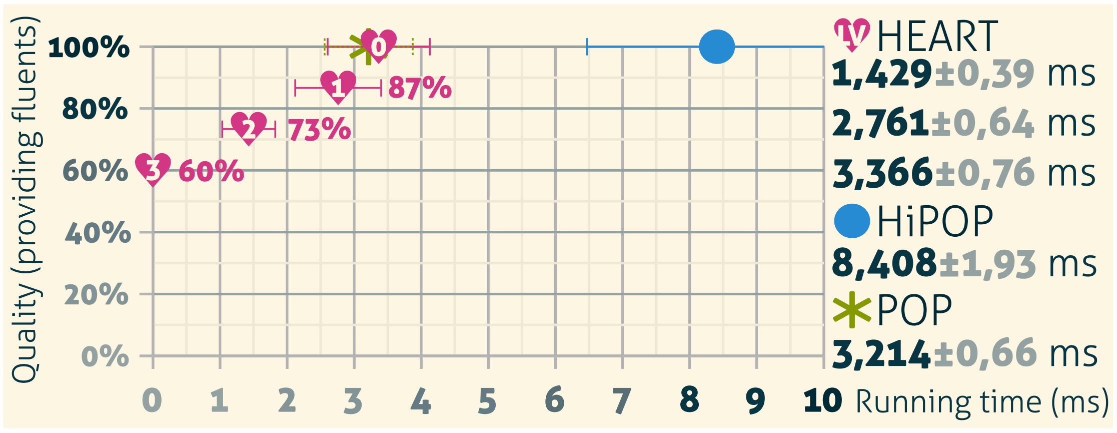


Figure 6.11: Evolution of the quality with computation time.

Figure 6.11 shows how the quality is affected by the abstraction in partial plans. The tests are made using our example domain (see listing ??). The quality is measured by counting the number of providing fluents in the plan $\left|\bigcup_{a \in S_\pi} \mathit{eff}(a)\right|$. This metric is actually used to approximate the probability of a goal given observations in intent recognition ($P(G|O)$ with noisy observations, see (Sohrabi *et al.* 2016)). The percentages are relative to the total number of unique fluents of the complete solution. These results show that in some cases it may be more interesting to plan in a leveled fashion to solve HTN problems. For the first cycle of level 3, the quality of the abstract plan is already of 60%. This is the quality of the exploitation of the plan *before any planning*. With almost three quarters of the final quality and less than half of the complete computation time, the result of the first cycle is a good quality/time compromise.

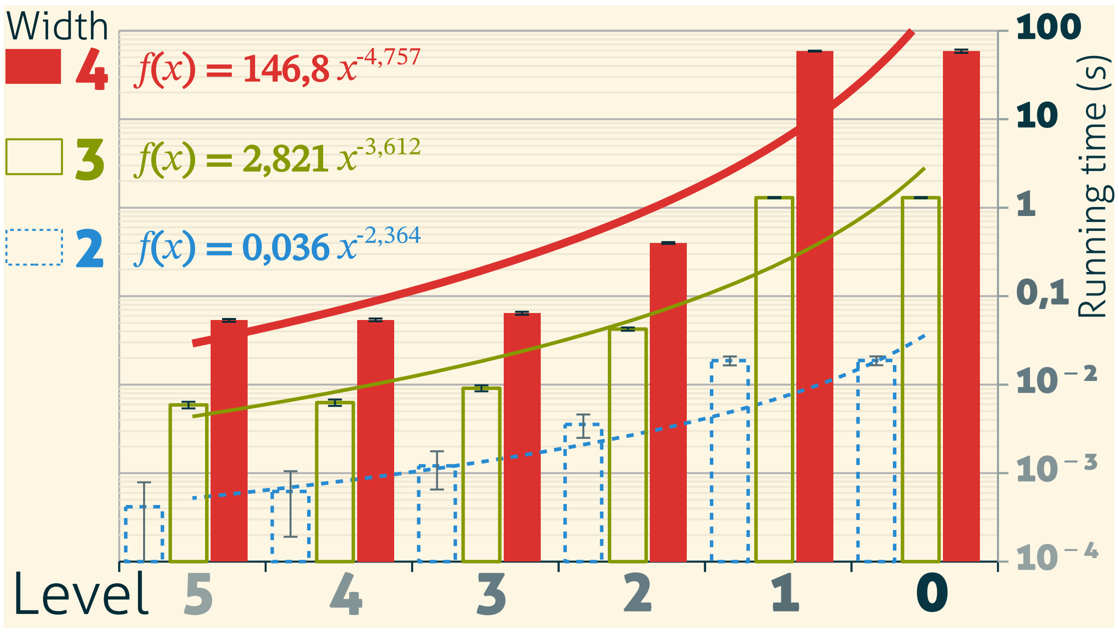


Figure 6.12: Impact of domain shape on the computation time by levels. The scale of the vertical axis is logarithmic. Equations are the definition of the trend curves.

In the second test, we used generated domains. These domains consist of an action of abstraction level 5. This action has a single method containing a number of actions of levels 4. We call this number the width of the domain. All needed actions are built recursively to form a tree shape. Atomic actions only have single fluent effects. The goal is the effect of the higher level action and the initial state is empty. These domains do not contain negative effects. Figure 6.12 shows the computational profile of HEART for various levels and widths. We note that the behavior of HEART seems to follow an exponential law with the negative exponent of the trend curves seemingly being correlated to the actual width. This means that computing the first cycles has a complexity that is close to being *linear* while computing the last cycles is of the same complexity as classical planning which is at least *P-SPACE* (depending on the expressivity of the domain) (Erol *et al.* 1995).

6.4 Conclusion

In this chapter we showed two planners that are oriented towards real time and flexibility. This makes fast decision making easier and improves the expressive power of domain writing tools such as the one we presented in chapter 5. oRiented

Such planners may be used in intent recognition using inverted planning. This technique is described in the next chapter.