Input control

In order to verify 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: $pre(a) = \bigcup_{a_s \in L^+(a)} causes(a_s)$ and $eff(a) = \bigcup_{a_s \in L^-(a)} causes(a_s)$. 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 and the domain compilation fails if POCL cannot be completed. Since our decomposition hierarchy is acyclic ($a \notin A_a$, see definition 10) nested methods cannot contain their parent action as a step. ## Problem

Problem instances are often most simply described by two components: the initial state and the goal.

Definition 5 (Problem)**.** The planning problem is defined as a tuple $\mathcal{P} = \langle \mathcal{D}, C_\mathcal{P}, a_0 \rangle$ where:

- $\mathcal{D}$ is a planning domain.
- $C_\mathcal{P}$ is the set of **problem constants** disjoint from the domain constants.
- a_0 is the **root operator** of the problem which methods are potential solutions of the problem.

Example: We use a simple problem for our example domain. The initial state provides that nothing is ready, taken or hot and all containers are empty (all using quantifiers). The goal is to have tea made. For reference, listing 2 contains the problem instance we use as an example.

```
1 init eff (hot(~), taken(~), placed(~), ~
      in ~);
2 goal pre (hot(water), tea in cup, water
      in cup, placed(spoon), placed(cup));
```

Listing 2: Example of a problem instance for the kitchen domain.

The root operator is initialized to $a_0 = \langle "", s_0, s^*, \{\pi_{lv(a_0)}\} \rangle$, with s_0 being the initial state and s^* the goal specification. The method $\pi_{lv(a_0)}$ is a partial order plan with the initial and goal steps linked together via a_0. The initial partial order plan is $\pi_{lv(a_0)} = (\{I, G\}, \{I \xrightarrow{s_0} a_0 \xrightarrow{s^*} G\})$, with $I = \langle "init", \emptyset, s_0, \emptyset \rangle$ and $G = \langle "goal", s^*, \emptyset, \emptyset \rangle$.

Partial Order Causal Links

Our method is based on the classical POCL algorithm. It works by refining a partial plan into a solution by recursively removing all of its flaws.

Definition 6 (Flaws)**.** Flaws have a *proper fluent* f and a causing step often called the *needer* a_n. Flaws in a partial plan are either:

- **Subgoals**, *open conditions* that are yet to be supported by another step a_n often called *provider*. We note subgoals $\ddagger_f a_n$ (see ??).
- **Threats**, 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 $\neg f \in eff(a_b) \wedge a_b \not\succ a_p \wedge a_n \not\succ a_b$. Said otherwise, the breaker can cancel an effect of a providing step a_p, before it gets used by its needer a_n. We note threats $a_b \otimes l_t$.

Example: Our initial plan contains two unsupported subgoals: one to make the tea ready and another to put sugar in it. In this case, the needer is the goal step and the proper fluents are each of its preconditions.

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

Definition 7 (Resolvers)**.** Classical resolvers are additional causal links that aim to fix a flaw.

- *For subgoals*, the resolvers are a set of potential causal links containing the proper fluent f in their causes while taking the needer step a_n as their target and a **provider** step a_p as their source.
- *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.

Example: The subgoal for $in(water, cup)$, in our example, can be solved by using the action $pour(water, cup)$ as the source of a causal link carrying the proper fluent as its only cause.

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.

Definition 8 (Side effects)**.** Flaws that are caused by the application of a resolver are called *related flaws*. They are inserted into the *agenda*[2] 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.

Example: Adding the action $pour(water, cup)$ causes a related subgoal for each of the preconditions of the action which are: the cup and the water must be taken and water must not already be in the cup.

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

In algorithm 1 we present a generic version of POCL inspired by Ghallab *et al.* (2004, sec. 5.4.2).

Algorithm 1 Partial Order Planner

```
1  function POCL(Agenda a, Problem P)
2      if a = ∅ then                    ▷ Populated agenda of flaws needs to be provided
3          return Success               ▷ Stops all recursion
4      Flaw f ← choose(a)               ▷ Heuristically chosen flaw removed from agenda
5      Resolvers R ← solve(f, P)
6      for all r ∈ R do                 ▷ Non-deterministic choice operator
7          apply(r, π)                  ▷ Apply resolver to partial plan
8          Agenda a' ← update(a)
9          if POCL(a', P) = Success then   ▷ Refining recursively
10             return Success
11         revert(r, π)                 ▷ Failure, undo resolver application
12     a ← a ∪ {f}                      ▷ Flaw was not resolved
13     return Failure                   ▷ Revert to last non-deterministic choice
```

For our version of POCL we follow a refinement procedure that works in several generic steps. In figure 1 we detail the resolution of a subgoal as done in the algorithm 1.

The first is the search for resolver. 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.

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 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.

In definition 8, we mentioned effects that aren't present in classical POCL, namely *negative resolvers*. All classical resolvers only add steps and causal links to the partial plan. Our method needs to remove composite steps and their adjacent links when expanding them.

The Heart of the Method

In this section, we explain how our method combines POCL with HTN planning and how they are used to generate intermediary abstract plans.

Figure 1: Example of the refinement process for subgoal resolution

Additional Notions

In order to properly introduce the changes made for using HTN domains in POCL, we need to define a few notions.

Transposition is needed to define decomposition.

Definition 9 (Transposition)**.** In order to transpose the causal links of an action a' with the ones of an existing step a in a plan π, we use the following operation :

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

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: $a \rhd^- a'$ gives all incoming links of a with the target set to a' instead.

Definition 10 (Proper Actions)**.** Proper actions are actions that are "contained" within an entity. We note $A_a = A_a^{lv(a)}$ 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: 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 11 (Abstraction Level)**.** This is a measure of the maximum amount of abstraction an entity can express:[3]

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

Example: The abstraction level of any atomic action is 0 while it is 2 for the composite action $make(drink)$. The example domain (in listing 1) has an abstraction level of 3.

Abstraction In POCL

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 1 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. We also need to add a way to deal with composite actions once inserted in the plan to reduce them to their atomic steps.

Definition 12 (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*: An decomposition flaw is solved with an **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: $a_n\oplus_\pi^m = \langle S_m \cup (S_\pi \setminus \{a\}), a_n \triangleright^- I_m \cup a_n \triangleright^+ G_m \cup (L_\pi \setminus L_\pi(a_n))$.
- *Side effects*: An decomposition flaw can be created by the introduction of a composite action in the plan by any resolver and invalidated by its removal :

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

Example: When adding the step $make(tea)$ in the plan to solve the subgoal that needs tea being made, we also introduce an decomposition flaw that will need this composite step replaced by its method using a decomposition resolver. In order to the decomposition of a composite action into a plan, all existing links are reported 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.

[3]We use Iverson brackets here, see notations in table 1.

Cycles

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 13 (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: In our case using the method of intent recognition of 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.

Example: In the figure 2, we illustrate the way our example problem 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.

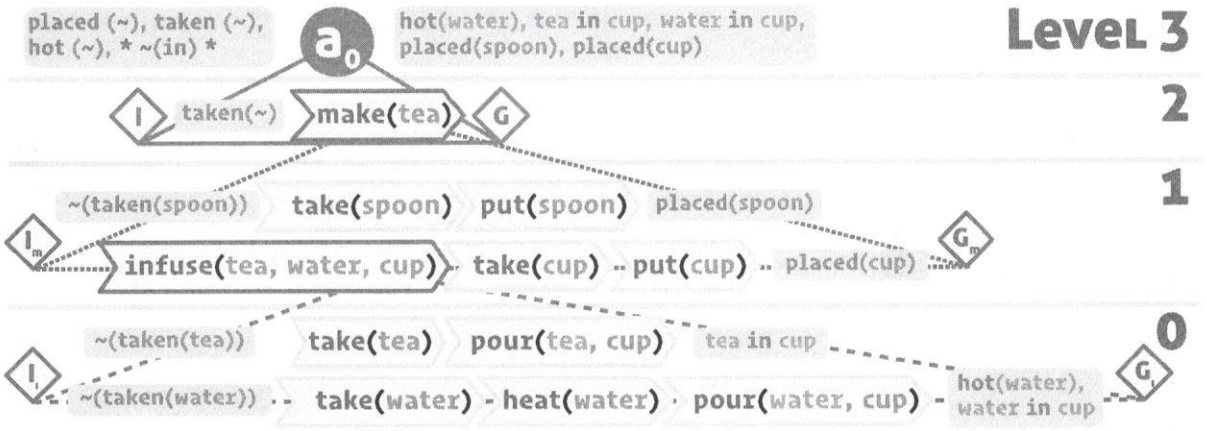

Figure 2: The cycle process on the example domain. Atomic actions that are copied from a cycle to the next are omitted.

Results

Theoretical

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

The completeness of POCL has been proven in the same paper as for its soundness (Penberthy *et al.* 1992). An interesting property of the POCL algorithm 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 (even if it is not needed given the soon to be proven properties).

Lemma (Decomposing preserves acyclicity). *The decomposition of a composite action with a valid method in an acyclical plan will not cause cycles to appear. This can be written* $\forall a_s \in S_\pi : a_s \not\succ_\pi a_s \implies \forall a'_s \in S_{a\oplus_\pi^m} : a'_s \not\succ_{a\oplus_\pi^m} a'_s$.

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 $\forall a_s \in S_m : (\nexists a_t \in S_m : a_s \succ a_t \wedge \neg f \in \mathit{eff}(a_t)) \implies f \in \mathit{eff}(a)$. 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 action 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 $\exists a \in \pi \implies S_m \cup S_\pi$ 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 4): $a \notin A_a$. 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 11 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 $a \notin A_{a\oplus_\pi^m}$ (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 $a\oplus_\pi^m$, 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 actions of the domain).

Experimental

In order to assess its capabilities, HEART was tested 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.[4] Each experiment was repeated between 700 and 10 000 times to ensure that variations in speed were not impacting the results.

Figure 3: Evolution of the quality with computation time.

Figure 3 shows how the quality is affected by the abstraction in partial plans. The tests are made using our example domain (see listing 1). The quality is measured by counting the number of providing fluents in the plan $\left|\bigcup_{a \in S_\pi} eff(a)\right|$. This metric is actually used to approximate the probability of a goal given observations in intent recognition. 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.

In the second test, we used generated domains. These domains consist of an action of abstraction level 5. It has a single method containing a number of actions of level 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. ?? 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 details of the formalism) (Erol *et al.* 1995).

[4]The source code of HEART will be available at gemi.io/heart

Conclusions

In this paper, we have presented a new planner called HEART based on POCL. An updated planning framework fitting the need for such a new approach was proposed. We showed how HEART performs compared to complete planners in terms of speed and quality. While the abstract plans generated during the planning process are not complete solutions, they are exponentially faster to generate while retaining significant quality over the final plans. They are also a proof of solvability of the problem. By using these plans it is possible to find good approximations to intractable problems within tight time constraints.

References

Baioletti, M., S. Marcugini, and A. Milani
Encoding planning constraints into partial order planning domains, *International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann Publishers Inc., 1998, 608–616.

Baker, C. L., J. B. Tenenbaum, and R. R. Saxe
Goal inference as inverse planning, *Proceedings of the Annual Meeting of the Cognitive Science Society*, Vol. 29, 2007.

Bechon, P., M. Barbier, G. Infantes, C. Lesire, and V. Vidal
HiPOP: Hierarchical Partial-Order Planning, *European Starting AI Researcher Symposium*, IOS Press, 2014, 264,51–60.

Beckett, D., and T. Berners-Lee
Turtle - Terse RDF Triple Language, W3C Team Submission W3C, March 2011.

Bercher, P., S. Keen, and S. Biundo
Hybrid planning heuristics based on task decomposition graphs, *Seventh Annual Symposium on Combinatorial Search*, 2014.

Biundo, S., and B. Schattenberg
From abstract crisis to concrete relief preliminary report on flexible integration on nonlinear and hierarchical planning, *Proceedings of the European Conference on Planning*, 2001.

Chen, J., Y. Chen, Y. Xu, R. Huang, and Z. Chen
A Planning Approach to the Recognition of Multiple Goals, *International Journal of Intelligent Systems*, 28 (3), 203–216, 2013. doi:10.1002/int.21565.