Programmatic Task Network Planning

Felix Mohr and **Theodor Lettmann** and **Eyke Hüllermeier** and **Marcel Wever**

{firstname.lastname}@upb.de
Paderborn University
Warburgerstrae 100
33098 Paderborn

Abstract

Many planning problems benefit from extensions of classical planning formalisms and modeling techniques, or even require such extensions. Alternatives such as functional STRIPS or planning modulo theories have therefore been proposed in the past. Somewhat surprisingly, corresponding extensions are not available for hierarchical planning, despite their potential usefulness in applications like automated service composition. In this paper, we present programmatic task networks (PTN), a formalism that extends classical HTN planning in three ways. First, we allow both operations and methods to have outputs instead of only inputs. Second, formulas may contain interpreted terms, in particular interpreted predicates, which are evaluated by a theory realized in an external library. Third, PTN planning allows for a second type of tasks, called oracle tasks, which are not resolved by the planner itself but by external libraries. For the purpose of illustration and evaluation, the approach is applied to a real-world use case in the field of automated service composition.

Introduction

It has been known for a long time that defining a planning problem often means to *strategically* organize the search space instead of only describing what is possible. In the initial version of PDDL, this was called the "advise" facet of the definition as opposed to the "physics", which are neutral and only describe what is possible in a domain. And as anticipated, new paradigms such as functional STRIPS (Geffner 2000), numerical planning (Hoffmann 2003), and, more recently, planning modulo theories (Gregory et al. 2012) and planning with the creation of constants (Weber 2009) emerged and significantly improved the language expressivity, the solver efficiency or even both.

Unfortunately, these extensions have not (or only marginally) been transferred to hierarchical planning. One of the main areas of application of hierarchical planning is automated software composition, and specifically that planning domain would strongly benefit from these extensions. More precisely, we are interested in three extensions:

1. *Constant Creation*. Planning actions should be allowed to add new objects to the environment as in (Weber 2009) or (Mohr 2017).
2. *Interpreted Predicates*. Preconditions of methods should be allowed to contain predicates whose truth value can be *evaluated* using background theories (and the state); this is very similar to the PDDL extension proposed in (Gregory et al. 2012).
3. *Oracle Tasks*. For some tasks, it is cumbersome to model its possible refinements by traditional HTN methods, e.g., deciding how to partition a set. Oracle tasks are like primitive tasks that are not linked to operators but to an external function that computes its possible applications *itself*.

In parts, these features have been already implemented in the SHOP2 planning system (Nau et al. 2003). SHOP2 allows for so called *external calls*, which allow to invoke external routines, which, in a way, can be used to interpret predicates and resolve oracle tasks. However, the concrete abilities of SHOP2 in this aspect have not been documented or discussed in scientific literature, so its formal scope is somewhat unclear.

In this paper, we realize these three extensions for hierarchical planning and merge them into a framework we call programmatic task network (PTN) planning. While the first two aspects are rather a transfer of existing classical planning approaches to hierarchical planning, a mechanic like oracle tasks is, to the best of our knowledge, a novelty of our approach.

One question we are particularly after is whether these extensions yield practical advantages. That is, we want to see whether problems can (i) be expressed in a more compact way and (ii) be solved more efficiently in terms of runtime.

To this end, we present a case study in the area of automated machine learning. In fact, the idea of applying hierarchical planning to automated machine learning was our main motivation to extend classical HTN planning. While we do not claim that PTN is relevant for most or even all hierarchical planning domains, the formalism should be relevant also for other problems in the sub-field of automated service composition. Since the case study is a real-world example, a side-contribution of the paper is to demonstrate the application of AI planning to a real use case.

Motivation and Running Example

Automated Machine Learning

Our extension of HTN is mainly motivated by the idea of tackling *automated machine learning* (AutoML) as a planning problem. AutoML is a recent research direction in

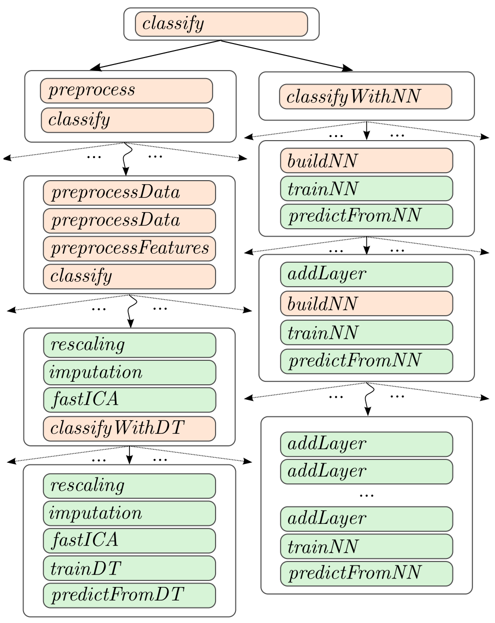

Figure 1: Task networks allow for composing pipelines in a flexible way, and for configuring their elements.

machine learning, which aims at (partly) automating the process of developing a machine learning solution specifically tailored for a concrete data set (Thornton et al. 2013; Feurer et al. 2015). Here, a solution is understood as an "ML pipeline", that is, the selection, composition, and parametrization of algorithms for training a predictor on the data. The latter includes, for example, methods for data pre- and post-processing, induction of a classifier, etc. The pipeline takes the data set as an input and produces a predictor as an output. The quality of a pipeline is measured in terms of the (estimated) generalization performance of the predictor, i.e., its predictive accuracy on new data. This quality may strongly vary between different pipelines.

Our idea is to build an ML pipeline with a hierarchical task network. The point of departure is a single task such as *classify*. This task can be refined recursively by iteratively prepending preprocessing steps and eventually choosing the concrete algorithms and their parametrization. For example, as shown in Figure 1, one could decide to have preprocessing steps before the classifier (left branch), i.e., the node *classify* is replaced by a sequence consisting of two nodes *preprocess* and *classify*. The node *preprocess* could then in turn be refined into two times *preprocessData* and once *preprocessFeatures*. Alternatively, it could be decided that no preprocessing is used (right branch).

Configuring Multi-Class Classifiers

For illustration, we consider a simplified version of the AutoML problem, in which we focus on the configuration of a single element of the pipeline, namely the classification algorithm, while ignoring other steps (such a pre- and post-processing). More specifically, consider the problem of

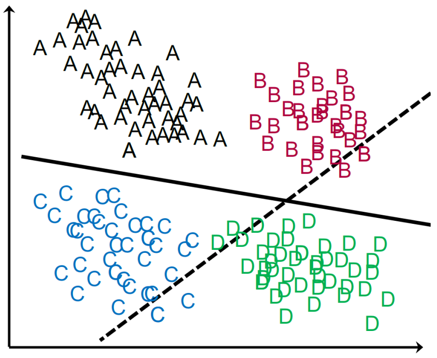

Figure 2: Classification problem with instances as points $x \in \mathbb{R}^2$ and four classes. The meta-class $\{A, B\}$ can easily be separated from $\{C, D\}$ by a linear classifier (solid line). Separating D from $\{A, B, C\}$ is more difficult (dashed line).

training a classifier $h : \mathcal{X} \longrightarrow \mathcal{Y}$, where $\mathcal{X}$ is an instance space (set of data objects) and $\mathcal{Y} = \{y_1, \ldots, y_K\}$ a set of $K > 2$ classes. So-called *decomposition techniques* reduce this problem to a set of binary classification problems, i.e., the training of a set of simple classifiers that can only distinguish between two classes. In so-called *nested dichotomies* (NDs), the reduction is achieved by recursively splitting the set of classes $\mathcal{Y}$ into two subsets (Frank and Kramer 2004).

Formally, a nested dichotomy can be represented by a binary tree, in which every node n is labeled with a set $c(n) \subseteq \mathcal{Y}$ of classes, such that the root is labeled with $\mathcal{Y}$, and $c(n) = c(n_1) \dot\cup c(n_2)$ for every inner node n with successors n_1 and n_2. Every inner node is associated with a binary classifier that seeks to discriminate between the "meta-classes" $c(n_1)$ and $c(n_2)$. At prediction time, a new object to be classified is submitted to the root and, at every inner node, sent to one of the successors by the binary classifier associated with that node; the class assigned is then given by the leaf node reached in the end.

A simple illustration with four classes is given in Fig. 2. Obviously, the dichotomy $((A, B), (C, D))$ would be a good choice for this problem, since all classification problems involved ($\{A, B\}$ versus $\{C, D\}$, A versus B, C versus D) can be solved quite accurately with a simple linear classifier. The dichotomy $((A, D), (B, C))$, on the other hand, would lead to rather poor performance, because the classifier in the root will make many mistakes ($\{A, D\}$ cannot easily be separated from $\{B, C\}$). The dichotomy $(A, (B, (C, D)))$ will produce a mediocre result.

In this paper, we configure nested dichotomies using hierarchical planning, assuming that the *base learner* for solving binary problems is a linear support vector machine. Even this reduced configuration problem is rather challenging. In fact, the problem of finding a dichotomy that is optimal for a given set of data and for a fixed base learner comes down to searching the space of all dichotomies, and the size of this space is $(2n - 3)!!$ for n classes (Frank and Kramer 2004).[1]

[1]Here, !! is the double factorial, not taking the factorial twice.

Configuring nested dichotomies hierarchically seems natural: Starting at the root, the splits are configured iteratively until every leaf node is labeled with exactly one class. In other words, a "complex" problem such as training a classifier for classes $\{A, B, C, D\}$ is (recursively) refined into simpler problems, such as training a classifier for classes $\{A, B\}$ and training a classifier for classes $\{C, D\}$ (and appropriately combining the two). Each of these problems again defines a classification task, which is solved in the same way (even though other techniques could be applied in principle).

The Baseline: The Classical HTN Formalization

We now explain how the configuration of such NDs can be encoded as a classical hierarchical planning problem. The formalization ensures that each ND is constructed exactly once. We need three operators, corresponding to primitive tasks:

1. $\mathit{init}(n, x, lc, rc, nc)$
   Pre: $x \in n \land \bullet(lc) \land \tau(lc, rc) \land \tau(lc, nc)$
   Post: $\bigwedge$
   $\mathit{true} \rightarrow x \in rc \land bst(x, rc) \land sst(x, rc)$
   $\forall x_n : x_n \in n \land x_n \neq x \rightarrow x_n \in lc$
   $\forall x_2, x_o : x \neq x_2 \land x_2 \in n \land sst(x, n) \land (x_o \notin n \lor x_o > x_2) \rightarrow sst(x_2, lc)$
   $\forall x_s : sst(x_s, n) \land x_s \neq x \rightarrow sst(x, lc)$
   $\neg \bullet (lc) \land \bullet(nc)$
2. $\mathit{shift}(y, x, l, r)$
   Pre: $x \in l \land bst(y, r)$
   Post: $x \in r \land bst(x, r) \land \neg x \in l \land \neg bst(y, r)$
3. $\mathit{close}(l, lw, r, rw)$
   Pre: $lw \in l \land rw \in r$
   Post: $\emptyset$

Intuitively, the idea behind these operators is to split up the labels of a node until every leaf node is labeled with a single class. A node is refined by creating two child nodes (via the init operator), where initially all classes except one (x) of the parent are in the left child. Then, we can use the shift operator to move single classes from the left to the right child. The predicates bst and sst are used to memorize the biggest and smallest elements of nodes, which is necessary to avoid mirroring NDs, i.e. one separating A, B from C, D and the other C, D from A, B The close operator can be used to guarantee the existence of at least one class in each of the children, which are the "witnesses" lw and rw; this guarantess soundness of solutions.

The relatively complicated notation of the $\bullet$ and τ predicates is to efficiently simulate the creation of objects. The idea is that there is a counter for the next newly created constant, which is shifted whenever an object is "created"; the state of this counter is maintained with $\bullet$. The initial state then needs to contain some successor chain $\tau(c_0, c_1), .., \tau(c_{n-1}, c_n)$ that indicates the order in which the constants are created. Here, n iconstants may be created (the $n + 1$-th constant cannot be created, because no successor is known for it). Hence, it is actually possible to simulate the creation of objects with the only limitation that some bound n for the number of such objects needs to be set. Previous approaches for simulating output creation (Klusch, Gerber, and Schmidt 2005) have used a different, simpler, encoding, which leads to a blow-up of the search space as analyzed in our experimental evaluation section.

While one may object that outputs are then only syntactical sugar, we would argue that a native support for outputs is quite desirable in both the problem formalization and the implementation of tools. On the theoretic side, allowing for outputs is naturally a good thing because this constitutes a specific planning problem, which is generally undecidable even without hierarchies (Hoffmann et al. 2009). On the practical side, planning is precisely about offering syntax (and semantics) to simplify the specification of a special kind of search problem. There are planning domains, in particular software configuration, where outputs are first-class citizens. In HTN planning, simulating the constant creation not only complicates the description of operations but also propagates to methods as can be seen below. Hence, outputs alone may not justify an *implementation* of an entirely new planner but motivate the support of outputs as part of the problem description.

We need two tasks with five methods to complete the specification. The first task is $\mathit{refine}(n)$, which means that the classes of node n shall be split up somehow. The second task is $\mathit{config}(l, r)$, which means that classes are to be moved from the left to the right child of some node. There are three methods for $\mathit{refine}(n)$:

1. $\mathit{finalSplit}(n, x, y, l, r, s)$
   Pre: $x \in n, y \in n, y > x, \bullet(l), \tau(l, r), \tau(r, s)$
   TN: $\mathit{init}(n, lc, rc, y)$
2. $\mathit{isolatingSplit}(n, x, l, r, s)$
   Pre: $x \in n, \bullet(l), \tau(l, r), \tau(r, s)$
   TN: $\mathit{init}(n, l, r, y) \rightarrow \mathit{refine}(l)$
3. $\mathit{doubleSplit}(n, x, y, l, r, s)$
   Pre: $x \in n, y \in n, y > x, \neg sst(x, n), \bullet(l), \tau(l, r), \tau(r, s)$
   TN: $\mathit{init}(n, l, r, y) \rightarrow \mathit{shift}(y, x, l, r) \rightarrow \mathit{config}(l, r) \rightarrow \mathit{refine}(l) \rightarrow \mathit{refine}(r)$

There are two methods for $\mathit{config}(l, r)$, which are

1. $\mathit{shiftElementAndConfigure}(l, r, x, y)$
   Pre: $x \in l, bst(y, r), x > y$
   TN: $shift(x, y, l, r) \rightarrow config(l, r)$
2. $\mathit{closeSetup}(l, lw, r, rw)$
   Pre: $lw \in l, rw \in r$
   TN: $\mathit{close}(l, lw, r, rw)$

The initial task network is then $\{\mathit{refine}(\mathit{root})\}$, where the initial state s_0 defines root and the ordering of classes. That is, $s_0 = \varphi(C) \land \bigwedge_{x \in C}(x \in \mathit{root})$, where C is the set of classes and φ maps C to an arbitrary explicit total order of items of C, e.g., the lexicographical order. The latter one is important to maintain the bst and sst predicates.

PTN Planning Formalism

Basic Planning Elements

As for any planning formalism, our basis is a logic language $\mathcal{L}$ and planning operators defined in terms of $\mathcal{L}$. The language $\mathcal{L}$ has first-order logic capacities, i.e., it defines an infinite set of variable names, constant names, predicate names, function names and quantifiers and connectors to build formulas. A *state* is a set of ground literals; i.e., it does not contain unquantified variable symbols. We do *not* adopt the closed-world assumption.

Like in planning modulo theories (Gregory et al. 2012), constants, functions, and a subset of the predicates of $\mathcal{L}$ are taken from a *theory*. A theory $\mathcal{T}$ defines constants, functions, and predicates and how these are to be interpreted. Predicates not contained in $\mathcal{T}$ behave like normal predicates in classical planning. That is, $\mathcal{L}$ consists of the elements of $\mathcal{T}$ together with uninterpreted predicates and constants. In the formalism, we use $\mathcal{T}$ as a formula itself.

An *operator* is a tuple $\langle name_o, I_o, O_o, P_o, E_o^+, E_o^- \rangle$ where $name_o$ is a name, I_o and O_o are parameter names described inputs and outputs, P_o is a formula from $\mathcal{L}$ constituting its preconditions and E_o^+ and E_o^- are sets of conditional statements $\alpha \rightarrow \beta$ where α is a formula over $\mathcal{L}$ conditioning the actual effect β, which is a set of literals from $\mathcal{L}$ to be added or removed. Free variables in P_o must be in I_o and free variables in E_o^+ and E_o^- must be in $I_o \cup O_o$.

The semantics of the planning domain are as follows. An *action* is an operator whose input and output variables have been replaced by constants; we denote P_a, E_a^+, and E_a^- as the respectively replaced preconditions and effects. An action a is *applicable* in a state s under theory $\mathcal{T}$ iff $s, \mathcal{T} \models P_a$ and if none of the output parameters of a is contained in s. Applying action a to state s changes the state in that, for all $\alpha \rightarrow \beta \in E_a^+$, β is added to s if $s, \mathcal{T} \models \alpha$; analogously, β is removed if such a rule is contained in E_a^-. A *plan* for state s_0 is a list of actions $\langle a_0, .., a_n \rangle$ where a_i is applicable and applied to s_i; here, s_{i+1} is obtained by applying a_i to s_i.

To summarize, the main difference in the basic planning formalism to classical planning is that operators have explicit outputs and that some predicates are not only evaluated from the state itself but the state together with some theory. Having theories available to evaluate expressions, predicates in the preconditions and effects may also contain terms others than simple variables. None of the two aspects is new by itself since output parameters have been considered in automated service composition previously (Weber 2009), and interpreted predicates have been considered prior to planning modulo theories (Gregory et al. 2012) through the notion of functional STRIPS (Geffner 2000).

Programmatic Task Networks

On top of this basic planning formalism, we now build a hierarchical model (Alford et al. 2016). A task network (HTN) is a partially ordered set T of tasks. A task $t(v_0, .., v_n)$ is a name with a list of parameters, which are variables or constants from $\mathcal{L}$. A task named by an operator is called *primitive*, otherwise it is *complex*. A task whose parameters are constants is ground.

The goal of HTN planning is to derive a plan for a given initial state and task network. That is, instead of reaching a goal state from the initial state (as in classical planning), we iteratively *refine* a given partial solution (the task network) until only primitive tasks are left.

While primitive tasks are realized canonically by an operation, complex tasks need to be decomposed by *methods*. A method $m = \langle name_m, t_m, I_m, O_m, P_m, T_m \rangle$ consists of its name, the (non-primitive) task t_m it refines, the input and output parameters I_m and O_m, a logic formula $P_m \in \mathcal{L}$ that constitutes the method's precondition, and a task network T_m that realizes the decomposition. The preconditions may, just as in the case of operations, contain interpreted predicates and functional symbols from the theory $\mathcal{T}$.

An method instantiation m is a method where inputs and outputs have been replaced by planning constants. m is *applicable* in a state s under theory $\mathcal{T}$ iff $s, \mathcal{T} \models P_m$ and if none of the output parameters of m is contained in s.

Leaving apart the different outputs of operations and methods and the functional elements in formulas, the definition of a PTN planning problem is analogous to the one of classical HTN planning. That is, a PTN planning problem is a tuple $\langle O, M, s_0, N \rangle$ where O is a set of operations as above, M is a set of methods, s_0 is the initial state, and N is a task network. The conditions for a plan $\pi = \langle a_1, .., a_n \rangle$ that is applicable in s_0 being a *solution* to a PTN problem $\langle O, M, s_0, N \rangle$ are inductive based on three cases:

1. N is empty. π is a solution if it is empty
2. N has a primitive task t without predecessor in N. π is a solution if a_1 realizes t and is applicable in s_0 and if $\langle a_2, .., a_n \rangle$ is a solution to $\langle O, M, \tau(s_0, a_1), N \setminus \{t\} \rangle$.
3. N has a complex task t without predecessor in N. π is a solution if there is an instantiation $\widehat{m}$ of a method $m \in M$ that is applicable in s_0 yielding a refined network N', and π is a solution to $\langle O, M, s_0, N' \rangle$.

Note that these cases are not mutually exclusive unless N is totally ordered.

The above HTN formalization extends classical HTN planning by object creation and interpreted predicates.

In PTN, we allow a third type of tasks we call *oracle* tasks. An oracle task t is a (primitive or complex) task that is associated with *functions* φ_t that generate *sets* of solutions (in the spirit of the above definition of a solution) to the subproblem $\langle O, M, s, \{t\} \rangle$. A programmatic task network is a hierarchical task network that may contain oracle tasks.

The notion of oracle tasks is an entirely algorithmic one and does not affect the semantic of the planning problem. The idea of oracle tasks is that the planner does not solve them by himself but *outsources* the computation of solutions to its oracle functions. In a sense, oracle tasks play the role of "complex" primitive tasks. They are primitive, because they are ground to actions within one planning step, but they are also complex, because they are not necessarily replaced by a single action but a sequence. However, with respect to the definition of a solution, oracle tasks are simply treated as primitive or complex, so whether a task is oracle does not affect the set of solutions to it.

Discussion

Prior to proceeding, we would like to discuss two aspect of PTN. First, what is the relation between using oracle tasks and interleaving planning and execution? Second, is using both interpreted predicates and oracles redundant?

The main difference between using oracle tasks and interleaving planning and execution is that the latter one is usually done to determine the successor state resulting from a *given* action. That is, the action that is going to be executed is *fixed* by the planner. Oracle tasks, in contrast, actually *outsource* a part of the planning (and decision) logic itself to the function bound to it.

The relation between interpreted predicates and oracles is complementary. On one hand, interpreted predicates are useful to ease the formalization process of a planning problem. On the other hand, oracle tasks aim at shortcutting some parts of the planning process and possibly prune alternatives. Of course, one can also achieve the same pruning effect using interpreted predicates, and this can sometimes make sense. However, we simply see the two concepts as used for different purposes and with different times of coming into action: Interpreted predicates enrich the formalization language, and oracles decrease the runtime of the planner. We also consider this aspect in our experimental evaluation.

Describing PTN Planning Problems

Unfortunately, there is no standard language to describe hierarchical planning problems as PDDL is for classical planning. One attempt to create such a language was made with ANML (Smith, Frank, and Cushing 2008), but it has not evolved to a standard. All existing HTN planners use their own format, so there is no commonly agreed point of reference which can serve as a basis for our extension. In a sense, the SHOP2 planner has created some kind of implicit proprietary standard (Nau et al. 2003). As a consequence, describing the way how PTN planning problems are described would come down to explaining the input syntax for our specific planner.

Hence, our format is proprietary, and we rather refer the reader to the technical documentation of the planner, which formally describes the syntax for describing the planning problems. In fact, the description language only puts a specific syntax for the formal items discussed above. The whole planning problem is defined in just one file with several sections for *types*, *constants*, *operations*, *methods*, and *oracles* respectively. There is no added value in describing this syntax in detail at this point.

However, we briefly want to discuss the definition of interpreted predicates and oracles since this is something technically new. As in (Gregory et al. 2012), interpreted terms are described in an extra file, one for each theory. Predicates that do not occur in any of these files are supposed to be not interpreted. Oracles are described by 6-tuples as follows:

```
[Oracles]
rpnd; refineND(n,lc,rc); n; lc,rc; card(n) > 1; rpnd.sh
cbnd; refineND(n,lc,rc); n; lc,rc; card(n) > 1; cbnd.sh
```

In this notation—where fields are separated by the ; symbol—, the first entry defines the name of the oracle, the second one the task addressed by the oracle followed by the input and output parameters, the precondition, and the external library that will be called to conduct the refinement. Note that our implementation is in Java and external libraries are either executable by the used operating system and invoked in a new process or Java classes implementing a specific interface, which, of course, is more performant.

We require that interpreted predicates are not only associated with an evaluation function but also with a *ground truth* function. For a given state, the ground truth function computes *all* possible groundings to objects of the state for which it evaluates to true. That is, a predicate cannot only be evaluated for a fixed ground parameters but they can even be *queried* for valid groundings. An example where this becomes important is the $ssubset$ predicate used in the following formalization; this predicate simply realizes the strict subset relation. The planner has not even information about possible *candidates* s for which it should evaluate $ssubsets(s, p)$, but the underlying set theory can *inspect* the object p in the state and generate objects representing the possible subsets.

In practice, it is not necessary to define methods for complex oracle tasks. The planner will not treat oracle tasks itself, so there is no reason to formalize the methods that can be used to refine it for the planner. In any case, the external libraries are specifically designed for a particular planning problem and usually only create valid solutions by construction. Since those libraries usually do not apply a planning algorithm themselves, a description of the available methods (or even the whole planning domain) can be omitted unless the libraries explicitly require those for whatever reason. Since PMT does not verify the correctness of the oracles' answers using the existing methods but simply trusts that they are correct, the formalization is optional and can be rather seen as a documentation.

We now explain how the nested dichotomy creation problem can be formalized as a PTN problem. We only need one primitive task (and operation) and one complex task (with two methods). The primitive task and its corresponding operation are responsible for configuring a specific split.

$config(p, s; lc, rc)$
Pre: $\emptyset$
Post: $\forall x : x \in s \rightarrow x \in lc, \forall x : x \in p \wedge x \notin s \rightarrow x \in rc$

The operation has two inputs and two outputs. The first input p is the node that is to be refined and the second one, s, corresponds to a specification of a subset of the elements of s that will appear in the left child. The outputs lc and rc are the data containers for the left and right child node respectively. Note that we do not need any precondition, because this action will only be used in a plan if other (method) preconditions were checked before. Since those method preconditions are sufficient, there is no need to formalize the actual preconditions of the action again. This effect of "shifted" preconditions is not special to our example but is common in HTN planning.

In addition to this operation, we need one complex task $refine(n)$ for which we have two methods:

1. $doRefine(n, s, lc, rc)$
   Pre: $\underline{ssubset}(s, p) \wedge\ !\overline{empty}(s) \wedge \underline{min}(s) = \underline{min}(p)$
   TN: $config(n, s, lc, rc) \rightarrow refine(lc) \rightarrow refine(rc)$
2. $closeNode(n)$
   Pre: $card(n) = 1$
   TN: $\emptyset$

The first method is to conduct an actual refinement of a node where the second is responsible only to detect that a node has already been refined to the end and can be closed. Once again, the comparison of the minimum element is needed to avoid so called mirrored dichotomies that are actually identical modulo switching the left and the right child of a node (see above formalization).

The preconditions of the methods only contain interpreted terms. All predicates are from a standard set theory as in (Gregory et al. 2012), so the dichotomy problem doesn't require a special theory. Note that, for the case of the first method, the strict subset condition plus the requirement that s is not empty implicitly requires that $card(p) \geq 2$.

In PTN-Plan, the complex task will be resolved using an oracle. Consider the precondition predicate $ssubset(s, p)$ and suppose there are no oracles for the task. When determining the applicable groundings of the operation, the planner must branch over all possible subsets s of p, i.e. $2^{|p|} - 1$ many candidates. This is usually infeasible even for very small p, because the planner must consider this exponential number of candidates not only once but also subsequently when analyzing possible successor nodes. Hence, PTN-Plan outsources the task grounding to an oracle task, which only produces a small number of these candidates. In our evaluation, we consider both cases to illustrate this effect.

A PTN Planner

We adopt a modification of forward decomposition (Ghallab, Nau, and Traverso 2004). In a nutshell, a rest problem in forward decomposition is a state together with a task network. Of course, initially, this is the initial state s_0 and the initially given task network N. Forward decomposition means to take one of the tasks in N that have no predecessors and resolve it either to an operation (if primitive) or to a new task network (if complex). Our planner, PTN-Plan, is written in Java. The implementation is available for public [2].

The classical forward decomposition algorithm must be modified in three ways. We discuss these modifications in detail in the subsequent sections.

Treating Output Variables

Even though outputs are motivated by operation outputs, the point where they become relevant in the algorithm are *methods*. While the constants are actually created by some action, methods need to talk over those outputs in order to establish a reasonable data flow in the task network they induce. For example, if we have a task `refineND(nd)` where `nd` is an object representing a nested dichotomy, we may have a method `configureAndRefineRecursively(nd,lc,rc)` with an induced totally ordered task network `crtAndConfig(nd,lc,rc) -> refineND(lc) -> refineND(rc)`, which is supposed to create two subsequent dichotomies `lc` and `rc` and distribute the elements of `nd` over them. So in fact, it is already clear at the method level that `lc` and `rc` will be produced elements and are not available yet.

[2]URL hidden during review phase

PTN-Plan stores the outputs of a method in opaque *data containers*. With respect to the planning formalism, data containers are nothing special but ordinary planning constants. Intuitively, data containers are what variable *names* are in typical imperative programming. That is, the container object itself is rather a reference to real semantic object than the object itself. PTN-Plan maintains a counter of newly created objects and labels them `newVar1`, `newVar2`, ... For this reason, it is forbidden to use constants with such a name in the problem description in order to avoid confusion.

In particular in the presence of theories, one may be interested in "complex" objects. In planning modulo theories, planning constants are not only some names but actually string representations of more complex objects such as a set. For example, the string "`{a, b, c}`" could be a planning constant with an intended meaning, which is obviously not known to the planner but only to the theory.

Even though PTN-Plan uses names for output objects instead of serializations according to some theory, more precise information about the container may become available later. The concrete value stored of data container, e.g., "`{a, b, c}`" will be determined by an action but usually not the method instance itself. So at the time of determining the method instance itself, it is not possible to say anything about the contents of a data container. But this is also not a problem, because the content description can be easily added using equality. For example, an operation can have an effect saying `o1 = union(i1,i2)` where `union` is a term from the set theory and `i1`,`i2`, and `o1` are inputs and outputs. Since `i1` and `i2` are known, the concrete serialization can be computed using the theory libraries as in (Gregory et al. 2012).

Treating Interpreted Terms

The point where interpreted terms become relevant to PTN-Plan is when it determines the method instantiations or actions that are applicable in a state. Both were defined to be applicable if their preconditions are satisfied in the state module the underlying theory $\mathcal{T}$.

Since the evaluation of interpreted predicates is potentially costly, PTN-Plan first evaluates the "normal" predicates. This process already implies a binding of most (and often all) of the variables of a method or operation, and the interpreted predicate has only to be checked for given parameters instead of determining for which of a given set of possible parameters it holds.

Unlike in planning modulo theories (Gregory et al. 2012), PTN-Plan does not evaluate interpreted terms. That is, PTN-Plan only distinguishes between predicates and terms but does not make a difference between primitive terms (constants) and complex ones (possibly nested expressions). In any case, both are simply string representations encoding some element of the respective theory and need to be decoded by the external library. At this point, we do not see any reason to have two different representations for the same constant within the planning calculus. Of course, PTN-Plan can be extended in this regard if we observe that collapsing a term to a simpler constant is useful in terms of runtime.

Technically, PTN-Plan supports the evaluation of interpreted predicates in two ways. First, PTN-Plan comes with a Java interface which can be implemented by a Java class used to evaluate a predicate. This is the most performant solution, because no new process needs to be spawned for the evaluation. For compatibility with external libraries, however, PTN-Plan also supports the call of stand-alone executable files. In that case, PTN-Plan expects the result of the evaluation (and nothing else) to be returned on the standard output stream; clearly, this variant is much slower.

Treating Oracle Tasks

When selecting an oracle task, PTN-Plan calls the respective oracle library and blocks until a set of solutions for the task arrives. As for interpreted terms, PTN-Plan supports Java oracle classes that implement a specific interface or external libraries that return the set of solutions (and nothing else) in a specific format over the output stream.

The oracle library is invoked with the reduced rest problem as its main parameter. The reduced rest problem is defined by the current state and the oracle task as the only task; the subsequent tasks are irrelevant.

Once the solutions have arrived, PTN-Plan creates one successor for each of the sub-solutions. The rest problem of those successors is the state that results from applying the respective sub-solution to the previous state, and the task network is simply the one of the previous rest problem without the resolved oracle task.

A subtle twist that has not been discussed so far is the fact that the oracle library may want to conduct an informed search, too. That is, PTN-Plan adopts a best-first search and uses some domain-specific source of information to compute the f-values of the nodes, and that source of information should be also available to the oracle libraries. However, this is no problem, because the common source of information can be stored as a resource, e.g., a file name, in a constant of the planning state. The oracle can then inspect that constant and acquire the desired information. In particular, no additional channel of communication is required.

A Brief Analysis of PTN-Plan

Correctness and Completeness

Assuming the correctness of solutions returned by oracles, the correctness of PTN-Plan is straight forward. The overall correctness of a solution π for the problem $\langle O, M, s_0, N\rangle$ follows from induction over the solution length n. PTN-Plan only returns an empty solution ($n = 0$) if $N = \emptyset$, which is correct. For $n > 0$, the first action a_1 of the solution is either inserted individually as the result of resolving a primitive task, or it is part of a sub-solution $\langle a_1, .., a_k\rangle$ generated by an oracle for a (complex) oracle task. The first case only occurs if PTN-Plan chose a primitive task $t \in N$ realized by a_1 and t is not preceeded by any other task in N; PTN-Plan only chooses a_1 if it is applicable. The second case only occurs if N has an oracle task not preceeded by any other task in N; the sub-solution $\langle a_1, ..a_k\rangle$ was then created by an oracle and is correct by assumption. In any case, the length of the solution to the rest problem is smaller than n; so the correctness of PTN-Plan follows from induction.

PTN-Plan is complete for problems without oracles or for oracles that create all solutions that exist for a single oracle task. This follows again by induction over the length of solution $\pi = \langle a_1, .., a_n\rangle$ for a problem $\langle O, M, s_0, N\rangle$. Three cases are possible. First, there is a primitive non-oracle task $t \in N$ realized by a_1 without predecessor in N; PTN-Plan considers a_1 as a possible refinement. Second, there is an oracle task $t \in N$ without predecessor to which $\langle a_1, .., a_l\rangle$ is a solution. Then, by assuming that oracles create all solutions, PTN-Plan obtains $\langle a_1, .., a_l\rangle$ from some oracle. Third, there is a complex non-oracle task $t \in N$ without predecessor in N. There is a refinement N' of N obtainable by the application of applicable method instantiations $m_1, .., m_k$ such that π is still a solution and that has no complex non-oracle task without predecessor in N'. But PTN-Plan considers these method instantiations in that order such that eventually one of the first two cases applies. In any case, PTN-Plan will eventually arive at a sub-solution $\langle a_1, .., a_l\rangle$ and a problem for which a solution $\langle a_{l+1}, .., a_n\rangle$ exists and which is found by the induction hypothesis.

However, since it is precisely the purpose of oracles to prune parts of the search space that seem irrelevant to them, PTN-Plan is not complete in general. That is, there are solutions that are not contained in the search graph of PTN-Plan. By the above analysis on completeness, this is the case if and only if the set of solutions created by the union of oracles defined for a task is a strict subset of the actual solution set. Consequently, oracle tasks can be used to trade completeness for search efficiency.

Heuristic Search

PTN-Plan adopts a best-first-epsilon algorithm to conduct the search over the graph induced by the planning problem. In our domains, A* is typically not applicable, because the criterion that is subject to optimization is not the plan length but some other qualitative property of the solutions that often does not decompose in an additive way over the edges of the search graph. For example, we cannot estimate the prediction accuracy of a nested dichotomy in an additive way over the search path.

As most hierarchical planners, PTN-Plan is not equipped with a built-in heuristic. We are aware of two planners with a built-in heuristic we are aware of. One is PANDA (Bercher and others 2015), and the other is Hierarchical Goal Network Planning (HGN) (Shivashankar et al. 2012; 2013; Shivashankar, Alford, and Aha 2017), which uses landmarks to compute a heuristic for the hierarchical planning problem. On the code level, PTN-Plan has an interface for the node evaluation function, which can be used to setup a problem-specific f-function. In principle, this f could also be additive, so the idea of PANDA could be used in PTN-Plan in principle if the actual cost measure is additive.

Experimental Evaluation

To assess the role of the different extensions to the overall performance, we compare not only HTN with PTN-Plan

Dataset	#	PTN	HTN + IP	HTN + C	HTN	PTN	HTN + IP	HTN +C	HTN
car	4	0,72	3,11 ●	4,47 ●	4,38 ●	22,00	75,78 ●	113,20 ●	98,94 ●
page-blocks	5	5,50	23,00 ●	39,50 ●	30,67 ●	52,00	326,50 ●	743,00 ●	522,00 ●
analcat	6	2,32	9,94 ●	22,89 ●	22,47 ●	68,91	307,29 ●	561,33 ●	539,42 ●
segment	7	6,74	79,84 ●	46,12 ●	47,47 ●	89,52	1.327,05 ●	891,29 ●	891,29 ●
zoo	7	1,79	1,65	3,22 ●	2,92 ●	72,07	146,35 ●	106,67 ●	87,38 ●
autoUni	8	6,26	69,25 ●	28,21 ●	26,73 ●	91,83	817,60 ●	205,53 ●	194,27 ●
cnae9	9	277,18	-	-	-	111,82	-	-	-
mfeat-fourier	10	25,24	-	117,18 ●	123,87 ●	121,56	-	335,91 ●	338,43 ●
optdigits	10	36,82	-	167,68 ●	165,85 ●	95,18	-	173,32 ●	178,25 ●
pendigits	10	16,33	-	93,92 ●	92,88 ●	102,92	-	346,92 ●	347,33 ●
yeast	10	8,83	137,50 ●	21,00 ●	20,67 ●	153,67	1.361,50 ●	295,80 ●	265,00 ●
vowel	11	3,44	-	31,50 ●	22,14 ●	111,11	-	994,83 ●	361,71 ●
audiology	24	11,78	-	118,69 ●	116,21 ●	235,50	-	849,85 ●	740,05 ●
letter	26	41,32	-	-	-	219,40	-	-	-
kropt	28	141,55	-	-	-	405,00	-	-	-

Table 1: Comparison of PTN-Plan with other extensions of HTN planning. The column entitled with # shows the number of classes for the respective dataset. The left main column reports the average runtime to the first solution in seconds. The second main column reports the number of nodes generated. A hyphen means that no solution was found in the timeout.

but also with other variants. More precisely, we consider the version of HTN but with constant creation or efficient creation simulation (HTN + CC) and the version of HTN with both constant creation and interpreted predicates (HTN + IP). That is, PTN and HTN + IP use the above formalization that adopts interpreted predicates; PTN outsources the *refine* task to an oracle (described below). HTN and HTN + CC use the initial formalization without interpreted predicates. In the case of plain HTN, the generation of objects is simulated naively and *without* the encoding shown in the first formalization; a formalization like this was used in (Klusch, Gerber, and Schmidt 2005).

The BF-ε search is informed by a simple f-function that completes the partial dichotomies using a technique called RPND (Leathart, Pfahringer, and Frank 2016) and then computes the performance of that dichotomy. Note that this f is not optimistic but rarely overestimates the optimal cost by large margin. The ε is considered as an absolute value (instead of a relative one) of 1% accuracy tolerance. The same technique is used by the oracles to generate a moderate number of possible refinements.

Our evaluation is based on a couple of datasets of different numbers of classes. This is because the search graph structure and size highly depends on the number of classes. In the general HTN encoding, the search graph size grows in a factorial order with the number of classes. Note that even if we use oracles and their massive pruning, the search space still grows quite rapidly simply because more decisions must be made in total; in fact, even the run time of a hill climber increases as least linearly. The datasets are from a well-known repository called UCI (Asuncion and Newman 2007) and are used frequently to evaluate the performance of algorithms that create nested dichotomies (Leathart, Pfahringer, and Frank 2016). All the datasets are available at `http://openml.org`.

Planning for AutoML pipelines is afflicted much more by random effects than planning in other domains due to intrinsic randomized aspects. The main sources of randomness are the *splits* made on the given data set. That is, to check the quality of a solution, the data set is initially split into two parts, the so called training set, which is used to guide the search, and test set, which is used to evaluate the quality of a solution (by comparing the dichotomy's prediction for each of these instances with the true class). In our experiments, this split is always 70%/30%. The choice of this split has paramount effects on the evaluation of the candidates, so it is necessary to consider not only one such split but several ones in order to get a more stable estimate. Also, the evaluation during search makes such splits which is why the evaluation of nodes is always subject to a certain degree of randomness. Since the search itself is not aware of the random nature of these values, it is important to obtain a relatively stable estimate for the mean, which is then the ultimate goal of optimization.

As a consequence, we report the mean values over 25 experiments for each data set. Table 1 shows the results of our computations. The computations were executed on 16 Linux machines in parallel, each of which with a resource limitation of 16 cores (Intel Xeon E5-2670, 2.6Ghz) and 16GB memory. Each experiment was conducted on a associated with a timeout of 5 minutes. We do not report the solution quality since this is not an important measure for the comparison of the search space exploration. However, we briefly summarize that PTN-Plan never produced significantly worse solutions than any of the other algorithms.

PTN-Plan clearly dominates each of the other algorithm variants in both runtime and node generation. Significant improvements (5%-threshold in t-test) are indicated by ●. For three of the problems, it is the only algorithm that identifies the solution within the given time bound. PTN-Plan generates only half the number of nodes as standard HTN on all dataset and sometimes 10 times less (page-blocks).

The improvement of PTN-Plan over HTN + IP motivates the use of oracle tasks in this context. Interpreted predicates allow for a very simple problem specification but yield an exponential number of successors for the refinement nodes, which frequently produces memory overflows. In PTN, only a small subset of those nodes is actually created, which makes it much more scalable.

These results, though rather preliminary, strongly motivate the usage of oracles in hierarchical planning. It is not at all clear that general purpose heuristics, despite all their advantages, can achieve the same performance as obtained using oracles (with very domain-specific heuristic knowledge). For the time-being, no such heuristics are in sight.

Conclusion

We have introduced an extension to classical HTN planning called PTN (Programmatic Task Networks) that connects the planner with external libraries in the form of logic theories and oracles. The theories are used to evaluate function terms and predicates that may occur in the preconditions of operations and methods. Oracles are used by the planner to outsource the generation of sub-solutions for specific tasks. We have conducted an experimental evaluation in the area of automated machine learning (AutoML), which was also our motivation to use (and extend) hierarchical planning. While the concrete problem of creating a nested dichotomy can also be solved easily without planning, HTN is a great framework to describe the construction mechanism of more general machine learning pipelines; PTN of HTN paves the way for a more efficient construction of those pipelines.

Open issues are on both theoretical and practical sides. Theoretically, it would be interesting to learn more about the possibility to transfer general heuristics from classical planning or HTN planning to PTN. On the practical side, PTN-Plan is still very preliminary and only supports totally ordered networks, so there is also a great deal of engineering research ahead.

Acknowledgements This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center “On-The-Fly Computing” (SFB 901).

References

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016. Hierarchical planning: Relating task and goal decomposition with task sharing. In *Proc. IJCAI*, 3022–3029.

Asuncion, A., and Newman, D. 2007. UCI machine learning repository.

Bercher, P., et al. 2015. Hybrid planning theoretical foundations and practical applications.

Feurer, M.; Klein, A.; Eggensperger, K.; Springenberg, J.; Blum, M.; and Hutter, F. 2015. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, 2962–2970.

Frank, E., and Kramer, S. 2004. Ensembles of nested dichotomies for multi-class problems. In *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004), Banff, Alberta, Canada, July 4-8, 2004*.

Geffner, H. 2000. Functional strips: a more flexible language for planning and problem solving. In *Logic-Based Artificial Intelligence*. Springer. 187–209.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated planning - theory and practice*. Elsevier.

Gregory, P.; Long, D.; Fox, M.; and Beck, J. C. 2012. Planning modulo theories: Extending the planning paradigm. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*.

Hoffmann, J.; Bertoli, P.; Helmert, M.; and Pistore, M. 2009. Message-based web service composition, integrity constraints, and planning under uncertainty: A new connection. *Journal of Artificial Intelligence Research* 35:49–117.

Hoffmann, J. 2003. The metric-ff planning system: Translating“ignoring delete lists”to numeric state variables. *Journal of Artificial Intelligence Research* 20:291–341.

Klusch, M.; Gerber, A.; and Schmidt, M. 2005. Semantic web service composition planning with owls-xplan. In *Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web*, 55–62.

Leathart, T.; Pfahringer, B.; and Frank, E. 2016. Building ensembles of adaptive nested dichotomies with random-pair selection. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, 179–194.

Mohr, F. 2017. *Towards automated service composition under quality constraints*. Ph.D. Dissertation, Paderborn University.

Nau, D. S.; Au, T.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: an HTN planning system. *J. Artif. Intell. Res. (JAIR)* 20:379–404.

Shivashankar, V.; Alford, R.; and Aha, D. W. 2017. Incorporating domain-independent planning heuristics in hierarchical planning. In *AAAI*, 3658–3664.

Shivashankar, V.; Kuter, U.; Nau, D. S.; and Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *Proc. AAMAS*, 981–988.

Shivashankar, V.; Alford, R.; Kuter, U.; and Nau, D. S. 2013. The godel planning system: A more perfect union of domain-independent and hierarchical planning. In *IJCAI*, 2380–2386.

Smith, D. E.; Frank, J.; and Cushing, W. 2008. The anml language. In *Proc. KEPS*.

Thornton, C.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2013. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA*, 847–855.

Weber, I. M. 2009. *Semantic Methods for Execution-level Business Process Modeling: Modeling Support Through Process Verification and Service Composition*. Springer.