Planning with Independent Task Networks

Felix Mohr, Theo Lettmann, Eyke Hüllermeier
{felix.mohr|lettmann|eyke}@upb.de

Department of Computer Science
Paderborn University, Germany

Abstract. Task networks are a powerful tool for AI planning. Classical approaches like forward STN planning and SHOP typically devise non-deterministic algorithms that can be operationalized using classical graph search techniques such as A*. For two reasons, however, this strategy is sometimes inefficient. First, identical tasks might be resolved several times within the search process, i.e., the same subproblem is solved repeatedly instead of being reused. Second, large parts of the search space might be redundant if some of the objects in the planning domain are substitutable.
In this paper, we present an extension of simple task networks that avoid these problems and enable a much more efficient planning process. Our main innovation is the creation of new constants during planning combined with AND-OR-graph search. To demonstrate the advantages of these techniques, we present a case study in the field of automated service composition, in which search space reductions of several magnitudes can be achieved.

1 Introduction

Hierarchical planning is an established and powerful technique for AI planning [1,3,13]. One interesting application of hierarchical planning is automated service composition, which is the task to compose a new software artifact from existing ones [8, 15, 19]. However, there are settings in which standard hierarchical planning, even when looking like a natural approach, turns out to be infeasible.

As an illustration, we consider the example of *nested dichotomies*, a technique for polychotomous classification in machine learning [5]. A nested dichotomy (ND) is a binary tree, in which every node n is labeled with a set $c(n) \subseteq \mathcal{Y}$ of *classes* $\mathcal{Y}$, 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. Fig. 1 shows two example dichotomies for the case of four classes. An 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. Since different NDs give rise to different sets of binary classification problems, the overall performance is strongly influenced by the tree topology. Considering an ND as a "classification service", hierarchical planning appears to be a natural approach for its configuration: starting at the root, the splits are configured iteratively until every leaf node is labeled with exactly one class.

The first problem with classical hierarchical planning is that, when enumerating different NDs, the same subsolutions are computed several times. For example, both

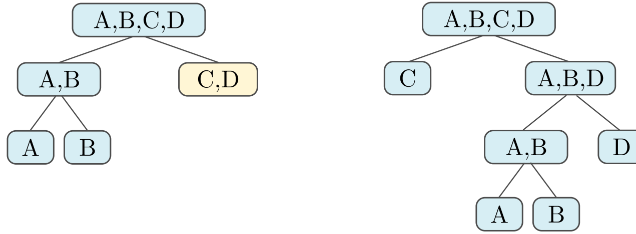


Fig. 1: A partial and a complete ND for four classes.

NDs in Fig. 1 contain the node A, B, which is refined twice by a classical planner. The second problem is that each node of the ND is represented by a planning constant, but the constants actually have no specific meaning. For example, we need 7 constants, say $v_1, .., v_7$, for the nodes of the right ND of Fig. 1. It does not matter which of the nodes is represented by which constant, but a classical planner tries all combinations, which yields an enormous and unnecessary search space explosion.

We propose planning with independent task networks (ITN) to overcome these problems (Section 2). The main novelties are the on-the-fly creation of planning constants and the reuse of subsolutions through the notion of AND-OR-graph search. In a case study, we show that this can yield search space reductions of several orders of magnitude (Section 3). The case study also sheds light onto a class of planning problems rarely considered in the planning community, e.g., the typical competitions, namely the one of automated service composition. While most frequently considered planning problems may not exhibit the discussed property of independent tasks, it *is* a common (sometimes essential) property in every recursive program. Drawing attention to this class of planning problems is, hence, another aim of the paper.

2 Independent Task Network Planning

We introduce our method in four steps. The first two subsections explain the formal basis of planning in general and hierarchical planning, respectively. We then describe the core elements of ITN planning and the ITN algorithm. Finally, we address some important aspects of the ITN that enable additional search space reductions.

2.1 Basic Elements of Planning

As for any planning formalism, our basis is a logic language $\mathcal{L}$ and planning operators that are defined in terms of $\mathcal{L}$. The language $\mathcal{L}$ has function-free first-order logic capacities, i.e., it defines an infinite set of variable names, constant names, predicate names, and quantifiers and connectors to build formulas. An *operator* is a tuple $\langle name_o, pre_o, post_o \rangle$ where $name_o$ is a name and pre_o and $post_o$ are formulas from $\mathcal{L}$ that constitute preconditions and postconditions, respectively.

The postconditions $post_o$ are often restricted to be literal sets, like in STRIPS. We relax this assumption a bit and allow conditional postconditions, i.e., $post_o$ is of the form $\bigwedge \alpha \rightarrow \beta$ where α is a formula from $\mathcal{L}$ and β is a set of literals.

A plan is a sequence of ground operations. As usual, we use the term *ground* to say that all variables have been replaced by terms that only consist of constants. That is, an operation is ground if all variables in the precondition and postcondition have been substituted by terms from $\mathcal{L}$ that only contain constants. Ground operators are also called *actions*; we write pre_a and $post_a$ for its precondition and postcondition, respectively.

The semantic of actions is that they modify the state in which they are applied. A *state* is a set of ground positive literals. Working under the closed world assumption, we assume that every ground literal not explicitly contained in a state is false. An action a is *applicable* in state s iff $s \models_{cwa} pre_a$. The *successor state* s' induced by this application is s if a is not applicable in s and $(s \cup add) \setminus del$ otherwise; here, add and del contain all the positive and negative literals, respectively, that are in a conditional postcondition of a whose condition is true in s.

2.2 Simple Task Networks

A simple task network (STN) is a partially ordered set T of tasks [7]. A task $t(v_0, .., v_n)$ is a name together 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.

We are interested in deriving a plan from a task network. Intuitively, we can refine (and ground) complex tasks iteratively until we reach a task network that only consists of ground primitive tasks, i.e., a set of partially ordered actions. While primitive tasks can be realized canonically by a single operation, complex tasks need to be decomposed by *methods*. A method $m = \langle name_m, task_m, pre_m, T_m \rangle$ consists of its name, the (non-primitive) task $task_m$ it refines, a logic formula $pre_m \in \mathcal{L}$ that constitutes a precondition, and a task network T_m that realizes the decomposition. Replacing complex tasks by the network specified in the methods we use to decompose them, we iteratively *derive* new task networks until we obtain one with ground primitive tasks (actions) only.

The definition of a simple task network planning problem is then straight forward. Given an initial state s_0 and a task network T_0, the planning problem is to derive a plan from T_0 that is applicable in s_0. A simple task network planning problem is then a tuple $\langle s_0, T_0, O, M \rangle$, where O and M are finite sets of operators and methods, respectively.

Note that the definition of a method usually contains more variables than the task it refines. That is, it makes use of objects that are not directly relevant for formulating the task, yet relevant to solve it in the spirit of the respective method.

2.3 Independent Task Networks: General Idea

We propose independent task networks (ITNs), which are an extension of STNs, with the purpose to enable an efficient decomposition of independent subproblems. The core feature of ITNs is that tasks may be labeled as *independent* to assert that each of its refinements is compatible with every refinement of non-preceding tasks. More formally, let T be a task network with $t \in T$ marked as independent, and let $T' \subset T \setminus \{t\}$ be the tasks in T that are no predecessors of t. Then for every state s on which we decompose

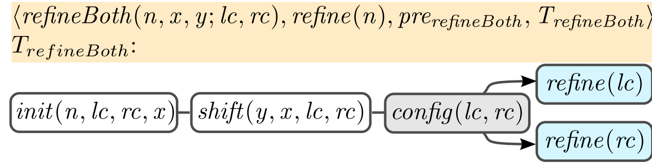


Fig. 2: The task network that refines node n of a partial nested dichotomy.

T, and for which plans π and π' can be derived from $\{t\}$ and T', respectively, such that $\pi.\pi'$ is applicable in s, *every* derivable plan π'' of $\{t\}$ applicable in s must be combinable with π' such that $\pi''.\pi'$ is applicable in s. In other words, the choice of the first partial plan π'' does not affect the applicability of the second partial plan π'.

As an example, consider the task network in Fig. 2. This is the task network belonging to the method that refines a non-terminal node of a partial dichotomy by splitting it up into two *new* child nodes. The first two tasks in the network are primitive, i.e., they can be realized by single actions, and the last three tasks are complex. The tasks $\mathit{init}(n, lc, rc, x)$, $\mathit{shift}(y, x, lc, rc)$, and $\mathit{config}(lc, rc)$ create the child nodes lc and rc of n and define their labels; the exact formalization is given below in Section 3.2. The tasks $\mathit{refine}(lc)$ and $\mathit{refine}(rc)$ mark a refinement of those child nodes, i.e., they are independent since their solutions are independent of each other. *Every* plan derived for $\mathit{refine}(lc)$ can be combined with *every* plan derived for $\mathit{refine}(rc)$.

The need to manually define whether or not a task is independent of the others has its root in the difficulty to define complete conditions of independence that can be checked automatically. Indeed, it is easy to specify sufficient conditions, e.g., based on the task names. For example, we can syntactically check whether two tasks must be independent if all methods and operators reachable from them have disjoint preconditions and effects respectively. However, this specific rule is too strict in general, and we expect that deciding independence in general is undecidable. On the other hand, in particular when the planning algorithm simulates a recursive execution tree—like in our nested dichotomy problem but also, the expert easily sees that the tasks are independent.

The non-deterministic independent forward decomposition (IFD) algorithm is shown in Algorithm 1. In fact, the part for $U = \emptyset$ is *equal* to the partial forward decomposition algorithm (PFD) [7][p.243] except that the recursive call is IFD and not PFD. So the important points are the computation of the relevant recursive tasks U in the beginning (line 1), and the final else-branch where those tasks are resolved (lines 16-19). Note that there is no choice point in the last branch, because all of the tasks must be solved—no decision is required. The independent tasks are solved in isolation and the solution of the remaining problem is appended to the concatenation of subsolutions of the independent tasks. It is easy to show that the routine is sound and complete; we omit the proofs of these formal properties due to space limitations.

A deterministic implementation of the above algorithm can be devised by an AND-OR-graph search such as general best first (GBF). As usual, the choice points (non-deterministic choices) constitute OR-nodes in such a graph. While PFD induces a simple OR-graph, the last branch of RFD induces an AND-node with one successor for

Algorithm 1: IFD(s, T, O, M)

```
1  if T = ∅ then return the emply plan U ← {t | t ∈ T, t has no non-recursive predecessor
   in T}
2  if U = ∅ then
3      choose any u ∈ T that has no predecessor in T
4      if u is a primitive task then
5          active ← {(a, σ) | a is a ground instance of an
6               operator in O, σ is a substitution such that
7               name_a = σ(t_u), and a is applicable to s}
8          if active = ∅ then return failure choose any (a, σ) ∈ active
           π ← IFD(γ(s, a), σ(T \ {u}), O, M)
9          if π = failure then return failure else return a.π
10     else
11         active ← {(m, σ) | a is a ground instance of a
12              method in M, σ is a substitution such that
13              name_m = σ(t_u), and m is applicable to s}
14         if active = ∅ then return failure choose any (m, σ) ∈ active
15         return IFD(s, σ(T_m), O, M)
16     end
17 else
18     ∀u ∈ U : π_u = IFD(s, {u}, O, M)}
19     π_{T−U} ← IFD(s, T \ U, O, M)
20     return π_{u_1}...π_{u_n}.π_{T−U}
21 end
```

each $u \in U$ and one for $T \setminus U$. Note that the child nodes here are partially ordered: there is no order among the child nodes for $u \in U$, but all of them are ordered previously to the node for $T \setminus U$.

2.4 A Look at the Details

While the RFD algorithm is already sufficient to solve subproblems independently, we allow for three more features that are important to actually achieve an efficiency improvement in the planning process. These features are *constant creation*, *context functions*, and *lonely methods*.

First, we allow operators (and methods) to introduce new constants. Intuitively, the connection to independent tasks is that those tasks constitute subproblems, which are *derived* from the current one. In Fig. 2, for example, refining the child nodes lc and rc are subproblems we derived from n. Since lc and rc are only relevant for this specific task, it is reasonable that they are created only for this purpose and known only within this method instead of being taken from a previously defined object storage. In the algorithm, this becomes relevant when active methods and actions are determined. Here, substitutions map output parameters not to constants of the state s but to globally unique new constants from $\mathcal{L}$.

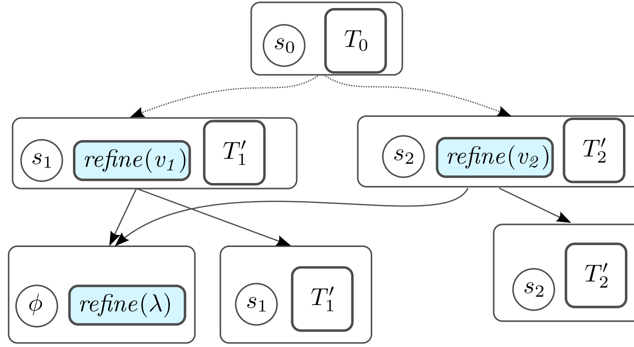


Fig. 3: Context functions help identify search graph nodes.

Second, ITNs allow one to equip tasks with context functions that enable the identification of equal subproblems *during the search process*. We face the problem that we may create independently solvable identical tasks that cannot be recognized as such. For example, the nested dichotomies in Fig. 1 both contain the node labeled A, B. Covering both dichotomies in the course of plan derivation, we would encounter a task $refine(v_1)$ for some state s_1 and $refine(v_2)$ for some state s_2, where v_1 and v_2 are different constants both encoding the refinement of A, B. That is, the subproblems $refine(v_1)$ and $refine(v_2)$ are identical, but $s_1 \neq s_2$ and $v_1 \neq v_2$ prevent us from detecting this equality. A context function $\phi_t : S \to S \times \Lambda$ overcomes this problem by assigning a state s a pair (s', λ), where $s' \subseteq s$ is a reduction of the state s, such that a plan derived from $\{t\}$ is applicable in s' iff it is applicable in s, and $\lambda \in \Lambda$ is a bijective mapping of constants in s' to constants in $\mathcal{L}$, i.e., a renaming of constants. In the above example, we would have $\lambda(v_1) = AB = \lambda(v_2)$, where AB is a constant, and the states are reduced to the literals really relevant to the subproblems such that $\phi_{refine(v_1)}(s_1) = \phi_{refine(v_2)}(s_2)$.

Incorporating context functions in the search algorithm simply means to change the recursive call for the tasks $u \in U$ in the lower else-branch to $RFD(\phi(s), \{\lambda(u)\}, O, M)$. Fig. 3 shows how this allows for the identification of nodes in the search space.

Third, methods may be declared as *lonely* in order to denote that the possible derivations of the resulting task network do not depend on the choices of the parameters. This is important for methods that only check some property while not affecting the state. For example, we may want to check that a node n is labeled with exactly one class. We achieve this by (i) choosing one class in the label of n, removing it, and (ii) checking whether the node label is then empty. If n would be labeled with several classes, the emptiness check (ii) would fail *independently* of which class we chose. Hence, we only need one representative of the labels of n to check the condition, i.e., only one instance of that method is required.

Just like simple task networks, ITNs induce a specific planning routine. It is common sense that simple task networks can only be reasonably solved through forward planning [7]. For ITNs, this is particularly true due to the context functions, which can not be evaluated until the state of invocation is known. Typically, this is only the case if the task network is resolved in a forward fashion.

3 Case Study: Configuration of Dichotomies

The improvements that can be achieved by ITN planning (in the settings where it applies) obviously depend on the concrete problem at hand. Here, we focus on the problem of ND configuration already presented in the introduction. For this example, we demonstrate a tremendous reduction of the search space (fully effective if AND-OR-graph search can reasonably be applied). Prior to proceeding, let us again emphasize that the approach is by no means restricted to this problem, but applies to other configuration problems (e.g., the configuration of deep neural networks) in very much the same way. All implementations are available for public[1].

3.1 Nested Dichotomies

As already explained, nested dichotomies reduce a polychotomous classification problem to a set of binary problems (that are presumably easier to solve). To this end, the set of classes is recursively partitioned into subsets, and for each such partition, a classifier is trained on a given set of training data. The criterion to be optimized is the overall prediction accuracy (percentage of correctly classified items), which depends on the quality of the binary classifiers, and therefore on the topology of the ND. Given a dichotomy, the accuracy can be estimated by training the required binary classifiers and applying the ND to suitable test data.

Since training and evaluation are not relevant here, we ignore these steps in our case study; instead, we focus on searching the space of nested dichotomies (topologies). This already constitutes a challenging planning problem. It has been shown that for n classes, there are $(2n-3)!!$ nested dichotomies [5], where !! is the double factorial (and *not* taking the factorial twice). Hence, for 10 classes, there are 34,459,425 many nested dichotomies—certainly too many for picking one by hand.

3.2 Problem Formalization

We now explain how the configuration of such NDs can be encoded as a hierarchical planning problem. The formalization makes sure that each ND is constructed exactly once. Besides the standard elements, it requires universal quantifiers, conditional postconditions whose conditions may be 2-CNFs, and *outputs*, which are separated by a semicolon. We need five operators, which will correspond to primitive tasks:

1. $init(n, x; lc, rc)$
   Pre: $in(x, n)$
   Post: $\bigwedge$
   $true \rightarrow in(x, rc) \wedge bst(x, rc) \wedge sst(x, rc)$
   $\forall x_n : in(x_n, n) \wedge x_n \neq x \rightarrow in(x_n, lc)$
   $\forall x_2, x_o : x \neq x_2 \wedge in(x_2, n) \wedge$
   $sst(x, n) \wedge (\neg in(x_o, n) \vee x_o > x_2) \rightarrow sst(x_2, lc)$
   $\forall x_s : sst(x_s, n) \wedge x_s \neq x \rightarrow sst(x, lc)$

[1] Sources are available at `http://www.felixmohr.de/en/research/crc901/itn`

2. $\mathit{shift}(y, x, lc, rc)$
   Pre: $in(x, l) \wedge bst(y, r)$
   Post: $in(x, r) \wedge bst(x, r) \wedge \neg in(x, l) \wedge \neg bst(y, r)$
3. $close(l, lw, r, rw)$
   Pre: $in(lw, l) \wedge in(rw, r)$
   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 smt 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 is used to guarantee the existence of at least one class in each of the children, which are the "witnesses" lw an rw.

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. In the following, lonely methods are annotated with an asterisk, and independent tasks are underlined. There are three methods for $\mathit{refine}(n)$:

1. $\mathit{finalSplit}^*(n, x, y; lc, rc)$
   Pre: $in(x, n) \wedge in(y, n) \wedge y > x \wedge \forall z : in(z, n) \rightarrow z = x \vee z = y$
   Task Network:
   $init(n, lc, rc, y)$
2. $isolatingSplit(n, x; lc, rc)$
   Pre: $in(x, n)$
   Task Network:
   $init(n, lc, rc, y) \rightarrow \underline{\mathit{refine}(lc)}$
3. $doubleSplit(n, x, y; lc, rc)$
   Pre: $in(x, n) \wedge in(y, n) \wedge y > x \wedge \neg sst(x, n)$
   Task Network:
   $init(n, lc, rc, y) \rightarrow \mathit{shift}(y, x, lc, rc) \rightarrow \mathit{config}(lc, rc) \rightarrow \underline{\mathit{refine}(lc)} \rightarrow \underline{\mathit{refine}(rc)}$

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

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

The initial task network is then simply $\{\mathit{refine}(root)\}$ where the initial state s_0 defines root and the ordering of classes. That is, $s_0 = \varphi(C) \wedge \bigwedge_{x \in C} in(x, root)$ 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.

3.3 Results

The evaluation is a mixture of experiments and *rough* bound estimates. On the one hand, it is non-trivial to calculate the exact search space sizes for a problem. Moreover, since the results cannot be immediately generalized, this calculation is not worth the effort. On the other hand, since we only want to demonstrate the general effects, namely orders of magnitudes of search space reduction, accurate values only distract from the key message. For the same reason we omit the proofs for the bounds. In fact, we determined better bounds than the ones we report here, but these are complicated to compute, which is not justified in light of the limitations imposed by the setup and space.

The results are summarized in Fig 4. In cases where the number could not be computed algorithmically, values with an asterisk were estimated based on expansion models. We now discuss the results in detail.

The Baseline: Standard STN/PFD Planning We can easily modify the above encoding to make it fit to standard STN planning. Since standard planning cannot create new objects, we must define a set of objects for the nodes of the dichotomy already in the initial state. Every nested dichotomy for k classes has $2k-1$ many nodes, one of which is the root, so the initial state of the problem must define the root node object and $2k-2$ additional node objects. The methods and operators that create objects are redefined in the sense that the outputs are now inputs. An auxiliary predicate $inuse(x)$, which is initially true for the root node and false for the other node objects, is required to be false in the preconditions of the "creating" methods and operator, and it is set to true in the postcondition of the creating operators. In addition, we add $lc \neq rc$ to the preconditions in order to make sure that the two "created" objects are distinct. Such a problem can then be fed to an implementation of PFD [7]; since we are interested in the total search space size, we used a simple breadth first search.

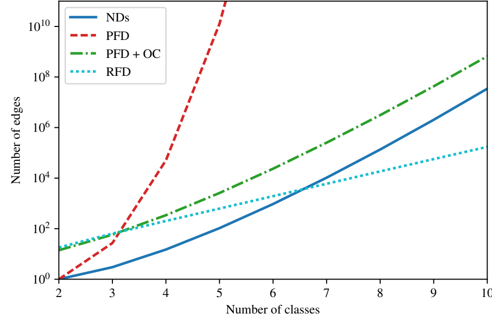
In principle, a more efficient encoding is possible for STN planning. When using alternative effects with universal quantifiers, we could simulate the constant generation process. However, these are not supported by common hierarchical planners, and such an encoding would also require a neat implementation of the planner in order to avoid an explosion of the node expansion time. Besides, this option is limited to cases where we already know the number of required constants, which is not the case in many scenarios, e.g., the configuration of a deep neural network.

The search space growth under this encoding renders the search process hopeless. One can show that the number of nodes induced for an OR-graph by PFD planning is at least $(2k-3)!!^k$, where k is the number of classes.

The extreme search space explosion is caused by an unnecessary redundancy in the set of found solutions. This is because different node objects are used to carry out the same operation. For example, a node n_i is split into children (n_{i+1}, n_{i+2}), (n_{i+1}, n_{i+3}), .., (n_{i+1}, n_l), .., (n_{l-1}, n_l), even though only one of those would be sufficient. This is avoided by creating new constants, which are built exactly for that single refinement purpose. This problem was discussed previously in the context of automated service composition [9]. As a consequence, PFD produces 2, 72, and 17 280 solution nodes for $k = 2, 3$, and 4, respectively, although there are actually only $1, 3$, and 15 distinct solutions.

# classes	PFD	PFD + OC	RFD
2	1	15	18
3	27	59	64
4	56 625	349	202
5	1.3E+10*	2 694	625
6	7.1E+17*	26 000	1 935
7	1.3E+28*	301 833	5 988
8	1.1E+41*	4 094 241	18 456
9	5.8E+56*	42 788 697*	56 563
10	2.4E+75*	660 099 747*	172 381

(a) Number of edges



(b) Number of dichotomies/edges.

Fig. 4: Search space sizes for the three models measured in terms of the number of edges, which corresponds to the number of nodes for STN. Values with asterisk were obtained by estimates, since the model exceeded the machine resources.

Improvement by Creating Constants Now consider the case that we still stick to an OR-graph search like PFD but allow the creation of new objects. That is, the encoding is as specified above, except that we apply PFD instead of the RFD algorithm introduced in Section 2.3. In the following, we call this strategy PFD + OC.

In comparison to the naive approach of a standard STN encoding, the search space size already looks much more feasible. For values of $k = 2, .., 10$, the values are contained in the second column of the table in Fig. 4. Clearly, the search space is still quite huge, i.e., still grows exponentially in the number of classes, but the order of magnitude is much less. More precisely, we can safely upper bound the search space size by $c \cdot k \cdot (2k-3)!!$, where k is the number of classes and c is a small constant. By the above lower bound for naive search, this implies that the search space size is smaller by a factor of at least $(\frac{2k-3!!}{c \cdot k})^{k-1} > (2k-3!!)^{k-2}$. This enormous gap can be observed in Fig. 4 between the green and the red line.

Improvement by ITN Planning Let us now consider the savings achieved by ITN planning. That is, we apply the RFD algorithm to the problem description as given above.

The result is again a dramatic search space reduction. The search space growth is still exponential but significantly less than in the case of STN planning with object creation. We can lower bound the search space size of PFD + OC by $k \cdot (2k-3)!! \gg 10^{k-2}$ and upper bound the search space size of ITN planning by 3^{k+1}. These bounds imply that the search space size of PFD + OC is at least 3^{k-2} times higher than the search space size induced by running an AND-OR-graph search on the graph imposed by the RFD algorithm. In other words, for deriving NDs, the search space of PFD + OC is exponentially larger than the one of ITN planning.

Another important (though maybe typical) observation one can make by comparing the two blue lines in Fig. 4 is that the number of edges in ITN planning grows *slower*

than the number of *solutions*. This is because the solutions are implicitly stored in the *sub-graphs* of the search space, so we actually need less nodes and edges to cover all solutions than in the other approaches. The impact of such an efficient representation can be quite paramount. For example, for the case of NDs it is often said that one cannot consider all NDs [5], which is a reasonable assertion at first sight given their tremendous number. Of course, there are limits. However, with an admissible and sufficiently informative heuristic for solution bases, we can actually (implicitly) consider all NDs even for sizes that significantly exceed the possibilities of OR-graphs.

3.4 Discussion

The case study of ND configuration impressively shows the potential benefits of ITN planning with respect to the search space size. In fact, the improvements are so obvious that no further discussion is needed. Instead, we dedicate the remaining space to the discussion of some more subtle aspects.

For example, a reduction of the search space does not immediately imply better solutions. First, in spite of all savings, we usually cannot construct the complete search graph. Instead, we still need to rely on heuristic search to explore promising parts of the search space. If these heuristics are good enough, it may happen that we find comparable solutions, or even the optimal ones, within a given time bound.

Second, an important requirement for successful AND-OR-graph search is that the quality of a solution can be aggregated from its partial solutions. If this is not directly possible, AND-OR-graph search may even deliver *worse* results than a simple best-first search, which has a complete solution base available in every node, no matter the search space size. However, at least for the shift from classical STN planning to STN planning with object creation, we *can* be certain that solution qualities will be at least as good and often better. Any heuristic we can apply for the classical STN planning version, we can also apply for the one with object creation. More precisely, for each node n of the search space of the object creation version, there is a *set* $N(n)$ of actually equivalent nodes in the search space of the classical problem formulation that are very likely to be *all* expanded before any solution is found.

To summarize, an ITN planning encoding does significantly decrease the search space size regardless of whether the search takes place in an OR-graph or an AND-OR-graph. Compared to the use of a classical encoding, this enables a much more efficient search. In this regard, AND-OR-graph search is even better than OR-graph search, but this approach assumes that solution quality can be aggregated from partial solutions.

4 Related Work

We are not the first in pointing out the necessity to create new constants during planning. In particular, for web service composition [10], the positive effect of allowing the introduction of new objects on the search space size was already discussed in [9]. In fact, such a technique was even incorporated earlier into a forward planning, [17], backward planning [12], and partial ordered planning [11]. However, we are not aware that constant creation has been used in hierarchical planning.

Constant creation can be simulated with effects that allow for negation, universal quantifiers, and implications. However, the only planners allowing universal quantifiers we are aware of, which are SIPE-2 [18], SHOP2 [13], and SIADEX [2], have no support for conditional effects; SHOP2 and SIADEX do not even support negations in the effects [6]. But in many cases, we have no canonical upper bound for these constants anyway. While we do have one in our example, in others, like configuring a deep neural network, there is no such bound for the number of layers.

A recent survey [6] categorizes HTN planning methods and discusses expressiveness of HTN planning languages and their impact on parallelizability. Currently, the most popular approach to implement HTN planning is depth first search in an OR-graph, which is adopted for instance by SIPE-2 [18], UCMP [3], SHOP2 [13], and SIADEX [2]. We are not aware of any other hierarchical planning algorithm that applies AND-OR-graph search.

The idea of reusing subsolutions has been addressed through the notion of "task sharing". Task sharing identifies common sub-tasks for sharing within a plan [16]. In [1] the HTN formalism is compared to a unified version of Hierarchical Goal Network (HGN) [14] and task sharing. However, task sharing only reuses subsolutions *within* a plan but does not use this knowledge within plan search, e.g., by organizing the search space like ITN.

5 Conclusion

We have introduced independent network planning as an alternative to classical hierarchical planning methods such as STN planning. While we do not claim that the required property of independent tasks is satisfied in planning problems frequently considered in the competitions (which it is probably not), we have shown at the example of nested dichotomy configuration that there *are* relevant practical problems where the conditions apply and where the search space size is decreased by several orders of magnitude. Nested dichotomies are not a pathological case: Since the core idea is to reuse computation results, we assume that ITN planning plays a role similar to dynamic programming, which makes it a key technology in automated service composition problem.

Our current work is focused on the use of ITN planning for automated machine learning [4], i.e., the automated configuration of data processing and model induction pipelines for learning predictive models from data. While our example of nested dichotomies originates from this domain, it constitutes only a first step and small share in this endeavor.

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

References

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

2. L. Castillo, J. Fdez-Olivares, Ó. García-Pérez, and F. Palao. Temporal enhancements of an HTN planner. In *Conference of the Spanish Association for Artificial Intelligence*, pages 429–438. Springer, 2005.
3. K. Erol, J. A. Hendler, and D. S. Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems, University of Chicago, Chicago, Illinois, USA, June 13-15, 1994*, pages 249–254, 1994.
4. M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.
5. E. Frank and S. Kramer. 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*, 2004.
6. I. Georgievski and M. Aiello. HTN planning: Overview, comparison, and beyond. *Artif. Intell.*, 222:124–156, 2015.
7. M. Ghallab, D. S. Nau, and P. Traverso. *Automated Planning - Theory and Practice*. Elsevier, 2004.
8. M. Klusch, A. Gerber, and M. Schmidt. Semantic web service composition planning with OWLS-XPlan. In *Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web*, pages 55–62, 2005.
9. F. Mohr. Issues of automated software composition in AI planning. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 895–898, 2014.
10. F. Mohr. *Automated Software and Service Composition - A Survey and Evaluating Review*. Springer Briefs in Computer Science. Springer, 2016.
11. F. Mohr. *Towards Automated Service Composition Under Quality Constraints*. PhD thesis, Paderborn University, 2017.
12. F. Mohr, A. Jungmann, and H. Kleine Büning. Automated online service composition. In *2015 IEEE International Conference on Services Computing, SCC*, pages 57–64, 2015.
13. D. S. Nau, T. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. SHOP2: an HTN planning system. *J. Artif. Intell. Res. (JAIR)*, 20:379–404, 2003.
14. V. Shivashankar, U. Kuter, D. S. Nau, and R. Alford. A hierarchical goal-based formalism and algorithm for single-agent planning. In *Proc. AAMAS*, pages 981–988, 2012.
15. E. Sirin, B. Parsia, D. Wu, J. A. Hendler, and D. S. Nau. HTN planning for web service composition using SHOP2. *J. Web Sem.*, 1(4):377–396, 2004.
16. D. E. Smith, J. Frank, and W. Cushing. The ANML language. In *Proc. KEPS*, 2008.
17. I. M. Weber. *Semantic Methods for Execution-level Business Process Modeling: Modeling Support Through Process Verification and Service Composition*, volume 40. Springer, 2009.
18. D. E. Wilkins. Can ai planners solve practical problems? *Computational intelligence*, 6(4):232–246, 1990.
19. D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S web services composition using SHOP2. In *International Semantic Web Conference*, pages 195–210. Springer, 2003.