Advances in Cognitive Systems 10 (2022) Submitted 9/2022; published 11/2022

Learning Decomposition Methods with Numeric Subtasks

Morgan Fine-Morris[1] MOF217@LEHIGH.EDU
Michael W. Floyd[2] MICHAEL.FLOYD@KNEXUSRESEARCH.COM
Bryan Auslander[2] BRYAN.AUSLANDER@KNEXUSRESEARCH.COM
Greg Pennisi[2] GREG.PENNISI@KNEXUSRESEARCH.COM
Kalyan Moy Gupta[2] KALYAN.GUPTA@KNEXUSRESEARCH.COM
Mark Roberts[3] MARK.ROBERTS@NRL.NAVY.MIL
Jeff Heflin[1] HEFLIN@CSE.LEHIGH.EDU
Héctor Muñoz-Avila[1] MUNOZ@CSE.LEHIGH.EDU

[1]Department of Computer Science and Engineering, Lehigh University, Bethlehem, PA 18015 USA

[2]Knexus Research Corporation, 174 Waterfront Street, Suite 310, National Harbor, MD 20745 USA

[3]Navy Center for Applied Research in AI, Naval Research Laboratory, Washington, DC 20375 USA

Abstract

We describe an HTN method-learning system, which we call T2N, that learns hierarchical structure from plan traces in domains with numeric effects and subgoals. The hierarchical structure is based on domain landmarks. We expect that a planner using a landmark-based decomposition hierarchy will be more time-efficient than a planner that uses an unbalanced right-recursive structure. We test learned methods on a numeric crafting domain and contrast the effectiveness of different decomposition structures, showing tradeoffs between structuring the decomposition hierarchy using landmarks or right recursion.

1. Introduction

Hierarchical Task Network (HTN) planners leverage user-provided domain information (called decomposition methods) to guide the process of generating a plan to accomplish a task or tasks. They decompose complex tasks, those not accomplishable by a single domain action, into progressively simpler ones to acquire a plan comprised of primitive tasks (accomplishable directly by domain actions). HTN planning can provide significant speed up over classical planning techniques, at the cost of extra up-front knowledge engineering required to define the decomposition methods (Ghallab et al., 2004; Nau et al., 2001). Many cognitive architectures, e.g., ICARUS (Langley et al., 2007), use hierarchical representations of plans, so learned hierarchies can benefit such architectures. A major problem in applying HTN planning to new domains is the expense and difficulty of authoring correct and useful decomposition methods.

A previous system (Fine-Morris et al., 2020) for learning decomposition methods for domains with numeric preconditions was not designed for domains where subtask decomposition depended

on numeric state variables. In this work, we describe a system (Trace2NumericHTN, abbreviated T2N) based on similar design principles, that can learn HTNs where changes in numeric variables dictate the structure of the network. We expect it to work for many hierarchical planning domains with symbolic effects, numeric effects, or a mixture.

In the current state-of-the-art of HTN learning, hierarchical structure is pre-determined. The resulting learned hierarchies are either binary decompositions and/or right-recursive decompositions, or the structure is inferred from provided domain knowledge, e.g., via user-defined skills as with ICARUS (Choi & Langley, 2005). T2N learns decomposition methods with numeric and symbolic preconditions and subtasks, utilizing a data-driven technique involving the clustering of word-embeddings on trace atoms (conditions or actions) to identify conditions that signal changes in the target subtask of a trace. Because the conditions bridge separate sub-problem contexts, these conditions are sometimes called bridge atoms (Gopalakrishnan et al., 2018; Fine-Morris et al., 2020). We use bridge atoms as "domain" landmarks, or landmarks that apply to more than one problem in a domain (hereafter we use bridge atom, landmark atom, and domain landmark interchangeably). These atoms then become the subtasks or subgoals into which our learned decomposition methods decompose the high-level tasks of a domain. The key difference between this work and that of Fine-Morris et al. (2020) is that these subgoals can include numeric variables, expanding the types of domains the learner can cover to include those where changes in numeric variables signal changes in sub-problem–i.e., domains where changes in numeric variables are important subgoals. This is useful for domains where action effects are both symbolic and numeric, but vital for domains where effects are exclusively numeric. We examine the performance of our system using a numeric crafting domain (Litecraft, based on the video game Minecraft) with exclusively-numeric effects. We learn several sets of decomposition methods and evaluate their performance in solving a set of randomly-generated test problems. Our results indicate that the selection of domain landmarks can impact planning efficiency and task coverage, but that the structures used in decomposing the primary subgoals also influenced planning efficiency.

The main contributions of this paper are:

- We learn HTN decomposition methods with numeric subgoals.
- We update our regression procedure so that it functions with numeric goals, by allowing some numeric precondition fluents to be ground to specific numeric values, inferred from initial states of the subtrace from which the method was learned.
- We introduce a new alternative "flat" decomposition structure for planning between landmarks, where a method for achieving a landmark can decompose into an arbitrary number of exclusively-primitive subtasks.
- In a domain with numeric subgoals, we perform a comparison of the trade-offs between (1) full right recursion as in HTN-Maker (Hogg et al., 2016); (2) landmarks with right recursion as in Fine-Morris et al. (2020); (3) the new decomposition structure using landmarks to decompose the top-level task and a flat hierarchy for planning between landmarks. Previously, we only examined methods with HTNs with structure matching (1) and (2).

- We collect statistics on each library of learned methods, comparing them based on the method count, the average method precondition count, and the average method subtask count.
- We add several new metrics to our analysis of method performance. We analyze the solutions to a set of test problems. Our analysis examines plan length, decomposition-tree depth, backtracking rate, and solving time.

2. Background

We will provide background information on HTN planning, planning and domain landmarks, extracting domain landmarks, and learning numeric preconditions over numeric actions using goal regression and function composition.

2.1 HTN Planning

An HTN planning problem, P, is a triple (D, s_0, T), where D is a domain description, s_0 is an initial state, and T is a list of tasks to accomplish. D is a tuple (O, M), where O is the set of operators and M are decomposition methods. An operator $o \in O$ is (h, p, e) and a method $m \in M$ is (h, p, $subs$), where h is the head (i.e., the task name and arguments), p are preconditions, e are effects, and $subs$ are a sequence of subtask heads. In section 5, we provide an example operator and example decompositions.

Preconditions are ($statevars$, $statements$), where $statevars$ are state variables and $statements$ are either (1) a calculation which stores its output to a temporary variable and (2) a comparison (e.g., $=, \neq, \leq, \geq$) of a variable and another variable or constant, where the variables may be either temporary or state variables. Temporary variables are used only within the scope of the method and are not state-variables. Effects are a set of pairs ($statevar$, $update$), where $statevar$ is a variable and $update$ is a constant or a temporary variable from $statements$ that is used to assign a new value to the $statevar$.

Each h of the operators and methods corresponds respectively to the primitive and complex tasks of the domain. In HTN planning, what constitutes 'accomplishing' a task is implicitly defined by the methods. In this work, we learn tasks concerned with achieving a goal (an atom). Therefore, we straddle two hierarchical planning paradigms, HTN planning and Hierarchical Goal Network (HGN) planning, which is similar to HTN planning but concerned with decomposing goals into subgoals instead of tasks into subtasks (Shivashankar et al., 2012).

2.2 Learning Numeric Preconditions with Function Composition

In addition to learning method subtasks, we also learn preconditions that may contain numeric functions. Traditional goal regression involves inverting each action, removing action effects, and introducing action preconditions. Fine-Morris et al. augments traditional goal regression to regress actions with numeric fluents via function composition. For example, when learning a method from a subtrace with initial state s_0 and actions a_1, a_2 with preconditions and effects:

$$p(a_1) = ([varX, \dots], [tmp1 \leftarrow f(varX, \dots),\ tmp1 > 1,\ \dots])$$
$$e(a_1) = [(varX, tmp1),\ \dots]$$
$$p(a_2) = ([varX, \dots], [tmp2 \leftarrow g(varX, \dots),\ tmp2 > 2,\ tmp3 \leftarrow h(varY, \dots),\ \dots]$$
$$e(a_2) = [(varX, tmp2),\ (varY, tmp3),\ \dots]$$

the method would inherit these precondition statements from the actions,

$$tmp1 \leftarrow f(varX, \dots),\ tmp1 > 1,$$
$$tmp2 \leftarrow g(tmp1, \dots),\ tmp2 > 2,\ tmp3 \leftarrow h(varY, \dots),\ varY \leftarrow tmp3.$$

Removing the tmp variables, the inequalities on $varX$ are equivalent to two conditions:

$$f(varX, \dots) > 1,\ g(f(varX, \dots), \dots) > 2.$$

The inequalities from both actions occur in the new set of preconditions, but in addition both $g(\dots)$ and $f(\dots)$ are applied to $varX$ before the inequalities from a_2 are checked, to ensure that whatever the value of $varX$, it will be greater than two after being updated according to both $f(\dots)$ and $g(\dots)$. We use a modified version of this goal regression technique.

2.3 Planning and Domain Landmarks

Planning landmarks are conditions that always occur in the solution of a given problem and have been addressed in many works (Hoffmann et al., 2004; Porteous et al., 2014). We are interested in finding what we call *domain landmarks*, common conditions for two or more problems in a domain, as opposed to *planning landmarks*, which are conditions common to one problem in a domain. Hereafter any reference to "landmarks" refer to domain landmarks instead of planning landmarks.

Given the set of all problems in a domain, Φ, we define a ϕ-domain landmark as a common planning landmark for every problem $P \in \phi$ where $\phi \subseteq \Phi$. Confirming a planning landmark for a problem is a PSPACE-complete problem (Hoffmann et al., 2004) and confirming a domain landmark will increase the running time by the factor $|\phi|$. Therefore, we offer a 'pragmatic' definition of ϕ-domain landmarks: any condition that occurs in at least one solution to every problem in ϕ. In our work, ϕ consists of problems that have a similar initial state s_0 and/or a list of tasks T, such that the problems will have common conditions for at least one solution for each problem in ϕ.

As in Fine-Morris et al. (2020), we use these domain landmarks as subtasks for the tasks demonstrated in the training traces. We also use these domain landmarks to partition the traces into subtraces from which we learn methods for accomplishing each domain landmark. Unlike Fine-Morris et al. (2020), in this work we learn a two-level hierarchy of methods. The landmark methods decompose single top-level tasks of the domain into subtasks based on the domain landmarks. These landmark tasks are decomposed by *subplan methods* into sequences of primitive tasks that achieve the landmark. For a generic example of hierarchies learned from a trace containing three landmarks, see Figure 1. A trace may not contain all selected domain landmarks, but as in Fine-Morris et al.

(2020), we partition traces at most once per landmark, so the maximum number of partitions per trace is $|LM|$ + 1 for a set of selected domain landmarks, LM.

Note that landmark methods derive their name from the use of domain landmarks to decompose the problem, **not** because they show the planner how to accomplish/achieve any individual landmark. They select which landmarks need to be achieved as subgoals of the final task. The role of determining *how* to achieve a landmark falls to the non-landmark methods which, unlike landmark methods, can have primitive subtasks.

2.4 Domain Landmark Extraction

We approximate the domain landmarks using a technique similar to the ones used in Gopalakrishnan et al. (2018) and Fine-Morris et al. (2020). This involves using the natural language processing algorithm Word2Vec (Mikolov et al., 2013) to learn a word embedding for each word (a condition atom or action) in the corpus (a set of traces). Word embeddings are vectors, each associated with a word, whose direction describes the context in which the word appears in the corpus. Therefore, two words with similar direction vectors can be assumed to co-occur frequently in the corpus. We use these vectors to determine how to split the words in the corpus into separate context-regions. We use Hierarchical Agglomerative Clustering with the cosine distance metric (a measure of the difference of the angle between two vectors) so that words that share contexts are clustered together. Because we expect that different contexts correspond to different domain sub-problems, the condition atoms that lie between different contexts can signal the end of one sub-problem and the beginning of a new subproblem. We can probe the boundaries between the clusters (context-regions) for atoms that signal a context shift. These condition atoms are our domain landmarks.

3. Learning Process

The learning problem is: given operator definitions, O, and final-task annotated traces, $\mathcal{T}$, obtain a set of subgoals, LM, and a library M of decomposition methods leveraging those subgoals to decompose the final-tasks of the training traces. T2N solves this learning problem in two phases: learning domain landmarks and learning methods. We briefly detail these phases and their steps. While not identical, the algorithm is very similar to the one described in pseudocode in Fine-Morris et al. (2020), and we describe our major modifications to that algorithm in section 3.3.

3.1 Phase 1: Learn Domain Landmarks

Phase one has three steps. Its inputs are a set of traces and its outputs are a set of domain landmarks. For a set of traces $\mathcal{T}$ (see section 6.1 for trace generation): Step (1.1) formats $\mathcal{T}$ and discretizes numeric fluents, Step (1.2) learns word embeddings for $\mathcal{T}$ using Word2Vec, and Step (1.3) selects bridge atoms that partition $t \in \mathcal{T}$ into subtraces.

Step 1.1: Preprocess Traces. Preprocessing comprises both linearization and skolemization. We linearized traces such that each condition or action in the trace can be treated as a word. For a trace $s_0, a_1, s_1, ..., s_N$, the linearized trace would be $c_{01}, ..., c_{0N}, a_1, c_{11}, ..., c_{1N}, ..., c_{N1}, ..., c_{NN}$, where each condition c_{XY} in the sequence $c_{X1}, ..., c_{XN}$ is true in state s_X. The sequence

$c_{X1}, ..., c_{XN}$ contains conditions of s_X that are in $e(a_X)$ or $p(a_{X+1})$ (where applicable). From a planning perspective, the ordering of the conditions within a state does not matter, and the conditions in a state can be reordered to create more input traces for Word2Vec. In a process similar to skolemization (Bundy & Wallen, 1984), we replace the numeric values with a string. Without this step, Word2Vec would interpret atoms that describe the value of the same variable as completely different words when they have even slightly different numeric values (e.g., `value(varX, 2)` and `value(varX, 3)`). This would increase the vocabulary size of the corpus and make it more difficult to learn relationships concerning numeric atoms.

Step 1.2: Learn Word Embeddings. We use Word2Vec with the skip-gram model to learn a set of word embeddings for the words in our corpus of linearized traces. The properties of Word2Vec ensure that words that occur in the same context are clustered by cosine distance. A small cosine distance for two word embeddings indicates that the corresponding words frequently co-occur.

Step 1.3: Select Landmark Atoms. The landmark learning process is as follows: we (1) learn word embeddings from the traces with Word2Vec, (2) form n=2 clusters of the word embeddings using a clustering algorithm and the cosine distance metric (we use Hierarchical Agglomerative Clustering), (3) score each atom according to the average cosine distance of the atom to those of the opposite cluster, (4) filter out non-effects atoms (i.e., any atom that is not an effect of an action), and (5) select atoms with scores less than $((max - min) \times .2) + min$ where max and min are the maximum and minimum scores of the effects atoms (i.e., any atom that is an effect of an action in at least one trace). The purpose of steps (2-3) are to split the atoms into at least two context groups and then find the atoms that are most in-between those two groups, i.e., the atoms in the corpus that have been assigned to one group but is as close as possible to another, possibly marking the boundary between the context groups. We choose two clusters semi-arbitrarily, as the best number of clusters will be domain-dependent and difficult to determine (we leave exploring this for future work).

3.2 Phase 2: Learn Methods

Phase two accepts the landmark atoms as input, and uses them to learn the decomposition methods that it outputs. It consists of three steps. Step (2.1) partitions $\mathcal{T}$ using the landmark atoms, Step (2.2) learns subplan methods in the flat or right-recursive style from each subtrace, and Step (2.3) learns landmark methods using the subplan methods. In purely binary right-recursive method libraries, we apply only Step (2.2).

During phase two, each trace is processed independently. Figure 1 shows how the two landmark-dependent decomposition styles can be learned from one trace (Figure 2 in section 5 shows Litecraft example decompositions and may be referred to for a more concrete example). The root contains the landmark method which has subtrees (some elided for space) for each of its four subtasks: one for each of the three landmark tasks plus the final task. In the flat hierarchy, the next level decomposes each landmark into a sequence of primitive tasks that accomplishes the landmark. In the right-recursive hierarchy, the next level is the root of a tree decomposing each landmark into a binary right-recursive structure. All leaves in the tree are primitive tasks.

Step 2.1: Partition Traces. T2N accepts as input a set of possible landmarks and a set of traces, each annotated with the associated task and outputs (subtrace, subtask) pairs for each processed

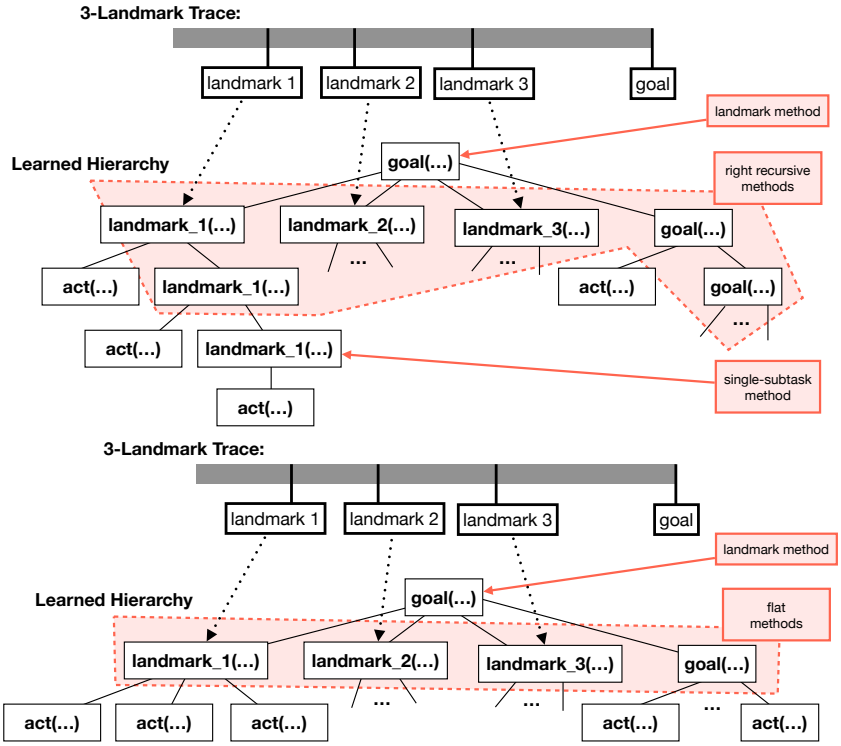

Figure 1. Examples of possible decomposition structures learned from a trace with three landmarks. The tasks are labeled with the type of method to which they correspond. The structure learned from a trace is determined by the number of landmarks in the trace and the number of actions in each subtrace. The top hierarchy combines landmark-based decomposition with binary right-recursion. The bottom hierarchy combines landmark-based decomposition with flat decomposition.

trace. Its discards any trace containing none of the landmarks because it is impossible to learn landmark methods from them, and filters traces to ensure that no two traces have the same sequence of actions. It partitions the traces on the first occurrences of each achieved landmark, for a total of at most $|LM| + 1$ partitions each terminated with a landmark or the final goal, as illustrated by the 3-landmark trace in Figure 1.

Step 2.2: Learn Subplan Methods. For each (subtrace, subtask) pair, T2N learns either one 'flat' method or a set of binary right-recursive methods, again illustrated in Figure 1, where the second row of the learned hierarchy represents the methods learned on each of the four partitions. The head of the subplan method(s) comes from the landmark atom that occurs in the final state of the subtrace. We learn preconditions for subplan methods via goal regression over the subplan.

Step 2.3: Learn Landmark Method. Once we have methods for each (subtrace, subtask) pair, we can learn a landmark method. The head is the final trace task, the subtasks are the landmarks plus the final trace task. The preconditions are learned via goal regression over the entire trace.

3.3 Updates to the Learning Technique

Our updates to the learning algorithm include modifications to precondition learning and changes in how tasks and subtasks are named.

Method preconditions are learned via goal-regression and function composition as described in section 2.2, however, we make two modifications to precondition learning by (1) grounding any numeric precondition fluents that are modified by the subplan but not already constrained by the preconditions and (2) merging pairs of methods whose preconditions are equivalent when they were also learned from equivalent subplans. Modification (1) allows us to constrain values of variables that are not already constrained by the preconditions learned via goal regression. This means that when variables are under-specified in the preconditions, we can learn additional information from the trace to improve planner behavior. To do this, we detect when a variable in the precondition $statevars$ is not constrained by the precondition $statements$, despite having a different value at the end of the section of trace we are learning from. We ground the value of that variable according to its value at the start of the trace. For example, if we are learning the method described in section 2.2 from the trace $s0\ a1\ s1\ a2\ s2$, the $statements$ learned via regression over $a1$ and $a2$ do not contain any direct condition statements on $varY$, but the effects of $a2$ update its value. The learner will detect this and prepend a new comparison to the $statements$, constraining $varY$ to value of $s0.varY$, the value of $varY$ in the initial state $s0$. Modification (2) allows us to combine sufficiently similar methods to avoid method duplication. For example, if $varY$ in the example from section 2.2 was ground to 1 in one method and 3 in an equivalent method, the merged method would constrain $varY$ to any value in the range $[1, 3]$. We attempt to find and merge equivalent methods for each method learned and keep only the merged method, discarding the originals.

In addition to the changes to goal regression, we also change task naming.Previously the final task of a trace was discovered by finding which of a set of user-provided possible final goals were present in the effects of the final action. In this work, each trace is annotated with a final task, which is not explicitly defined by a final goal condition, because this allows tasks to be named more flexibly. Previously, the landmark atom was used directly for each landmark task. Now, landmark atoms are converted to landmark tasks by transferring any non-variable arguments out of the task arguments and into the task name. E.g., `value(item, X)` becomes `value_X(item)` because `item` denotes a variable but `X`, which is a number or range of numbers, does not.

4. Example Domain: Litecraft

The domain used in this paper is a custom, numeric domain based on the crafting system of the game Minecraft. Agents gather base resources (wood, stone, coal, iron) from the world, and craft new items from them. The amount of a resource or item is described by atoms in the form `value(counter, amount)`, where `counter` is a label for a counter of a particular type of item, and `amount` is a numeric value. Actions update the state by increasing and decreasing counters associated with a particular stockpile of an item or resource, held either by the world or a particular agent.

There are two main categories of action: gather and craft. Gather actions are parameterized by agent, base resource type, and resource amount. Some resources require a tool to gather them. Each

Table 1. Crafting/Gathering Recipes.

craftable item	yield	station	ingredients	
wooden_plank	4		1 wood	
stick	4		2 wooden_plank	
crafting_table	1		4 wooden_plank	
wooden_axe or pickaxe	1	crafting_table	2 stick	3 wooden_plank
stone_axe or pickaxe	1	crafting_table	2 stick	3 cobblestone
furnace	1	crafting_table	8 cobblestone	
iron_ingot	1	furnace	1 coal	1 iron_ore
iron_axe or pickaxe	1	crafting_table	2 stick	3 iron_ingot
cart	1	crafting_table	5 iron_ingot	
rail	16	crafting_table	1 stick	6 iron_ingot

Table 2. Gather Recipes.

Gatherable Resource	Tool
wood	no tool or any axe
cobblestone	any pickaxe
coal	any pickaxe
iron (iron_ore)	stone pickaxe or iron pickaxe

tool type ({wood, stone, iron} $\times$ {axe, pickaxe}) corresponds to an action, and can only gather a resource if the tool type is appropriate for that resource (see Table 2 for resource requirements). Each gather action decreases the counter of the resource held by the world and increases the counter of the resource for the specified agent. Craft actions are parameterized by the agent. There is a craft action for each craftable item. Some craft actions require a station (a crafting_table for crafting complex items or a furnace for smelting ore into ingots) in addition to ingredients. Craft actions decrease the appropriate ingredient counters in the agent inventory and increase the product counter in the agent inventory by pre-determined amounts (see Table 1 for station dependencies, ingredient amounts, and yield amounts for each product). Neither stations nor tools are consumed when used to craft or gather (in Minecraft, tool durability decreases with use and the tool is consumed when durability=0, but this functionality is not implemented here). Table 3 shows a partial definition for the `CRAFT PLANK` action The preconditions specify that (1) the agent has a counter for a type of item in its inventory, (2) the counter is for wood, and (3) the counter has a value greater than or equal to one. Precondition (4) calculates (and stores to a temporary variable, `?tmp1`) the reduced amount of wood the agent has after one wood is consumed to make four planks. The effects update the value of the counter to the value of that temporary variable. Not shown is a similar update in the effects that would increase the counter of the planks by 4. Note that the ($statevar$, $update$) pair in the effects are written as an assignment instead of a tuple to emphasis that $update$ is used to change the value of the $statevar$.

Some items are more expensive to craft from scratch (i.e., starting with an empty inventory) than others because they require more steps. E.g., crafting a stone_pickaxe from scratch requires

Table 3. Example Litecraft Operator: Crafts Planks from Wood.

task	CRAFT PLANK(?agent)	
preconds	inventory(?agent_wood_counter, ?agent)	(1)
	item_type(?agent_wood_counter, WOOD)	(2)
	value(?agent_wood_counter) $>= 1$	(3)
	?tmp1 $\leftarrow$ value(?agent_wood_counter) -1	(4)
	...	
effects	value(?agent_wood_counter) $\leftarrow$?tmp1	
	...	

cobblestone, which must be gathered using a wooden_pickaxe. Therefore, the agent must craft a wooden_pickaxe *and* a stone_pickaxe. The items in Table 1 are ordered roughly by expense.

5. Sample HTN Structures

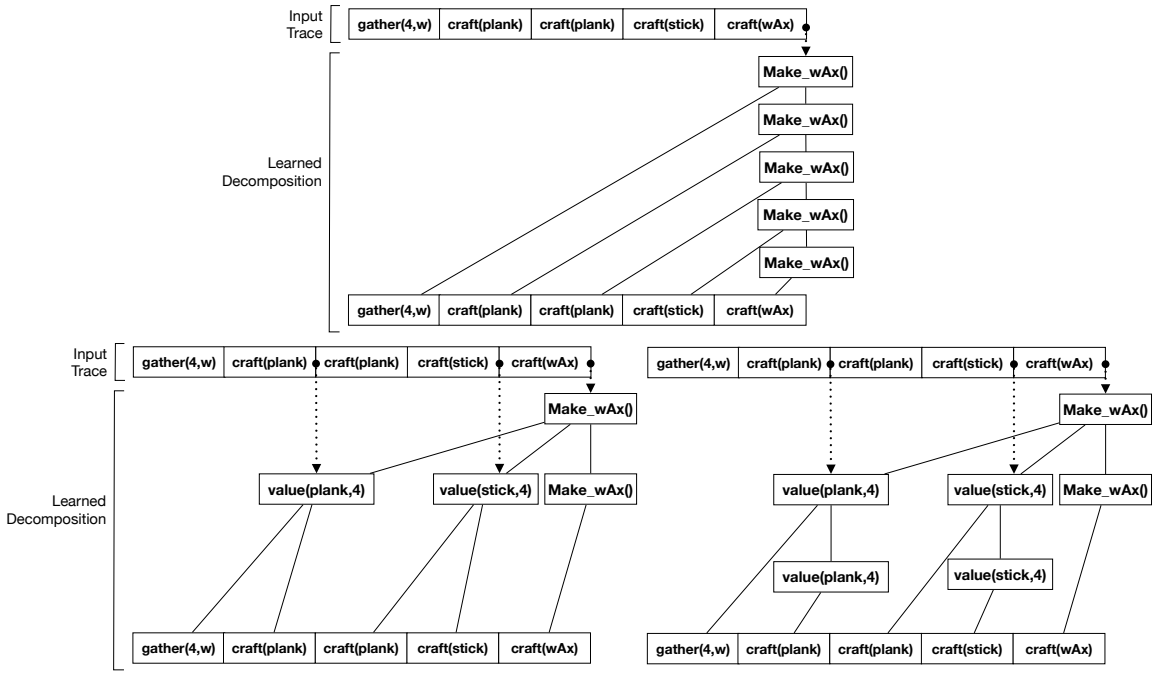

Figure 2. Three styles of decomposition. Purely right-recursive (top), landmarks with flat hierarchy (bottom left), and landmarks with right-recursion (bottom right). The *agent* parameter (which is normally the first parameter of each action) is omitted for space.

The three styles of decomposition learned in this paper are (1) a binary right-recursive (*brr*) style as learned by Hogg et al. (2016), (2) a style using landmarks and right-recursion, and (3) a style using landmarks to learn a flat hierarchy.

Examples of the three decomposition structures compared in this paper are displayed in Figure 2. Each decomposition is extracted from the same plan, which is displayed along the top of each diagram. The sequence of actions is: (1) gather 4 wood, (2) craft 4 planks from 1 wood, (3) craft 4 planks from 1 wood, (4) craft 4 sticks from 2 planks, (5) craft a wooden pickaxe from 3 planks and 2 sticks. For this example, we let the selected domain landmarks be `value(plank, X)`, `value(stick, X)` and the ground atoms in the traces are `value(plank, 4)` and `value(stick, 4)`. In each diagram, dotted lines drop down from the positions (between the actions) in the plan where the state containing each goal would occur. The task annotation for the plan is "Make Wooden Axe" (abbrev. as "Make_wAx"). This example trace assumes that the agent already has a crafting_table.

In the purely right-recursive decomposition (top), the final task of the trace is repeatedly decomposed by the learned methods into two subtasks: the first is a primitive subtask and the second is recursive on the final task. This continues until there is only one action left, at which point the final decomposition decomposes into the final action.

The second and third decompositions (bottom), which correspond to the generic hierarchies in Figure 1, use domain landmarks to partition each plan into three subplans, each terminated by the achievement of either a landmark or the final task. In each, the top-level task decomposes into 3 subgoals, (1) to acquire planks, (2) to acquire sticks, and (3) to recursively make a wooden axe. The second and third decompositions differ in how they decompose the three subgoals. In the second decomposition, each subgoal decomposes into the sequence of actions in the subtrace. In the third decomposition, each subgoal decomposes into a binary right-recursive structure (like the first decomposition): the left subtask is primitive and the right subtask is recursive, and this repeats until the final task of the recursion decomposes into a single subtask accomplished by the last action.

6. Experiments

All work for this paper was done on a 2020 MacBook Pro with a 2 GHz Quad-Core Intel Core i5 processor, 32 GB 3733 MHz LPDDR4X memory, running Big Sur (v. 11.6).

6.1 Training Problems and Solutions

We generate training traces by creating randomized training problems and solving them with a customized version of the Pyhop planner[1] with hand-crafted HTN methods, designed to produce plans with significant variation in subtask and action order which we believe helps the landmark learning procedure find true. As the plans are not annotated with decomposition information, we believe that any planner suffices.

The initial state of each training problem consists of counters for each gatherable item (wood, stone, iron, coal) indicating how many units of this resource are available for collection, and agent

1. https://bitbucket.org/dananau/pyhop/src/master/

counters for all items indicating how many of each the agent holds. The available-resource counters are randomized between 2000-5000 in steps of 100, and the agent's counters are initialized to zero. Tasks were of the form "make item" where the item is a craftable item from Table 1 where station=crafting table. Our hand-crafted methods prioritize plan variability over efficiency because we believe that it helps the landmark learner differentiate between landmarks that are incidentally vs. necessarily consistent across traces. In Litecraft, this meant that excess items could be gathered, and some tools could be crafted if they were useful even if they were not strictly necessary.

For each task, we generated 10 plans and 20 traces from each plan by reordering the conditions in each state randomly. Throughout this section and the results section, when we provide an average we follow it with the standard deviation. Traces contained an average of 14.8 ± 5.4 states, each comprising on average 16.2 ± 2.1 atoms. We annotated each training trace with the task to be learned.

6.2 Word2Vec Hyperparameters

We learn word embeddings using the following hyperparameters: alpha=0.001, min alpha=0.0001, epochs=1000. Vector dimensions were 11 (i.e., $floor(V/20)$ where V is the |corpus vocab|), and window size was 462 (i.e., $3C$, where C is the average |conditions| per state).

6.3 Test Problems

For each of the 9 final tasks of the domain, we generated 4 random solvable problems where the amounts of the gatherable resources were randomized between 200 and 5000 in steps of 10. The agent's inventory was empty (i.e., all counters held by the agent were initialized to zero). We confirmed problem-solvability by generating a plan using our customized Pyhop implementation and the same hand-written methods used to generate the training traces. We solved each test problem using one of the learned method libraries and the customized Pyhop that generated the training traces. It attempts methods in a consistent, deterministic order.

6.4 Metrics

To analyze our results for the test problems, we examine several metrics:

- **Task coverage** describes how many of the domain tasks are addressed by at least one landmark method, regardless of how successful the problems with that task are solved. If a library has no landmark methods for a particular task, it does not cover that task. If a library has landmark methods for a task, but cannot solve any of the test problems with that task in the time limit, it still covers that task. Task coverage applies only to libraries with landmark methods.
- **Problem coverage** measures how successfully a library solves test problems.
- **Backtracking rate** is the number of instances the planner was forced to backtrack during planning.
- **Plan length** is the length of each solution plan.

- **Planning duration** is the amount of time it takes for a planner to solve each test problem using a particular library.
- **Decomposition-tree depth** refers to the maximum height of the task decomposition tree constructed during planning. For the same task and initial state, a shallower depth indicates a more balanced tree.

We also compare statistics on the method libraries, including the number of methods, the average number of subtasks per method, the average number of total preconditions per method, and the average number of numeric expressions (conditions involving an inequality comparison or a calculation that saves its value to a temporary variable) in the method preconditions.

7. Results

In the results, we label the method libraries using acronyms. The purely right-recursive library (with no landmark methods) is labeled RR. The library with learned landmarks and a flat subgoal decomposition hierarchy is LL. The library with learned landmarks and right-recursive subgoal decomposition hierarchy is LLRR. The library with custom-selected landmarks and flat subgoal decomposition hierarchy is CL.

Task- and Problem- Coverage. Table 6 shows what percent of the nine final tasks each library covered. The CL library showed full task coverage and problem coverage. RR had full problem coverage. Task coverage was not applicable because the library was purely right recursive. Libraries LL and LLRR covered all tasks and problems except wooden_axe and wooden_pickaxe tasks. This shows a downside of learning landmarks using our current system: libraries constructed with learned landmarks may leave some tasks uncovered, as learned landmarks may not occur in all training traces.

Backtracking. Significant backtracking was not used by any method library, suggesting that learned method preconditions were usually thorough and sufficient. CL used backtracking once per iron_axe problem. The LL and LLRR libraries each used backtracking 7 times for each rail problem (which was also the task for which they took the longest time to solve). There was no backtracking for any other problems.

Plan Length. For all libraries, the average plan length was consistent within same-task problems. A task breakdown (Table 4) shows that for most final tasks, the plan length was the same for each library. The LL and LLRR libraries could not solve the wooden axe and pickaxe problems, so there is no data for those tasks. The library impacted plan length only for iron_axe & iron_pickaxe problems (18 for CL and 17 for the others) and rail problems (27 for LL and 20 for the others). Only for iron_axe, iron_pickaxe, and rail did the library impact the plan length. The similarities in length suggest that decomposition structure has little impact on plan length.

Decomposition-tree Depth. As expected for the flat-hierarchy CL and LL libraries, the mean decomposition-tree depth was 2.0 ± 0.0. For LLRR the mean was 9.0 ± 0.0 and for RR it was 12.6 ± 5.1. For a given (library, task) pair, the decomposition depth was always the same value, suggesting that the planner used the same sequence of methods per final task. The RR library was the only one to show final-task-dependent variation in depth because the depth was always equal to the plan length (see the RR column of Table 4).

Table 4. Plan Length.

Task Product	Library CL	LL	LLRR	RR
wooden (axe, pickaxe)	7	–	–	7
stone (axe, pickaxe)	9	9	9	9
furnace	9	9	9	9
iron (axe, pickaxe)	**18**	17	17	17
cart	18	18	18	18
rail	20	**27**	20	20

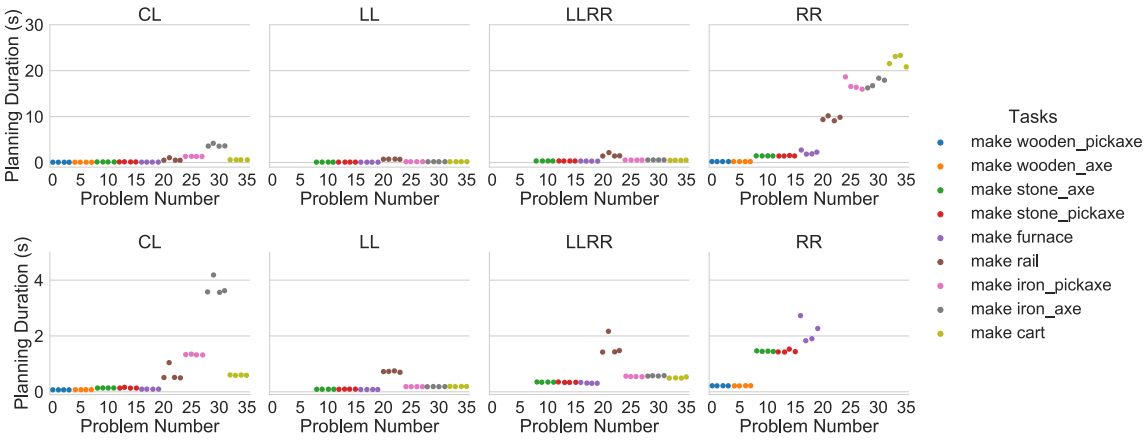

Figure 3. Time (in seconds) taken by an HTN planner to produce a plan using each method library, for each problem in a set of test problems. Both rows show the same dataset, but the bottom is zoomed-in to show the finer variation between zero and five seconds.

Planning Duration. Figure 3 shows the time for the planner to solve each test problem. Problems occur in the same order for all 4 graphs and are grouped by task (see legend). No library with landmark methods showed obviously superior performance across all task categories, however, for the tasks that it covers, LL is often the most efficient. For all but the easiest problems, the CL, LL, and LLRR libraries outperform the RR library.

None of the libraries required significant backtracking, so it is likely that the difference in solving times was due to library size, i.e., how many methods the planner had to check to find an applicable method. In RR, most methods were grouped into one of the nine final tasks, whereas in CL, LL, LLRR, methods were split into final tasks and landmark tasks, reducing the number of methods the planner checked to find an appropriate method.

Landmarks. Table 5 shows the landmarks used during method learning. The custom landmarks were selected to be intuitive–they involve acquiring tools and stations necessary for acquiring and crafting new resources. The learned set focuses on the furnace and cobblestone held by the agent, and the free resource counter for iron_ore. These are not particularly intuitive landmarks. Combined

with the comparative success of the LL and LLRR libraries compared with the CL library for metrics such as planning duration and plan length, this suggests that intuitive landmarks may not result in the most efficient method libraries.

Because so many of the plans were concerned with crafting items that required iron (axe, pickaxe, rail, cart) and stone (axe, pickaxe, furnace), the landmark learner prioritized atoms that divided those traces well, at the expense of the traces for crafting the easier wood items. Training sets biased to include more traces from more difficult problems in the domain (e.g., those requiring longer plans to solve) can help the landmark learner select landmarks that favor the more difficult problems. While this risks leaving some easier tasks uncovered by the landmark methods, these are also the problems most amenable to brute-force solutions, where the landmark methods are less necessary for quick solving time.

Table 5. Landmarks Used in Method Learning.

learned landmarks (LL & LLRR libraries)	hand-selected landmarks (CL library)
	`value(agent_crafting_table, X)`
`value(agent_cobblestone, X)`	`value(agent_wooden_pickaxe, X)`
`value(agent_furnace, X)`	`value(agent_stone_pickaxe, X)`
`value(available_iron_ore, X)`	`value(agent_furnace, X)`
	`value(agent_iron_pickaxe, X)`

Method Statistics. Table 6 shows statistics on the learned method libraries, with the operator statistics on the bottom row for comparison. The |preconditions| column shows the average total number of preconditions per method and the average number of numeric expressions in the preconditions. Numeric expressions comprise a large chunk of the total precondition count. As expected, LLRR and RR libraries are larger but have a smaller number of subtasks per method, while CL and LL are smaller but have a larger average subtask count.

Table 6. Operators and Learned-Method Library Statistics

Library	Task Coverage (%)	Size	Avg |Subtasks|	Avg |Preconditions| Total	Avg |Preconditions| Numeric Expr	Avg |Effects|
CL	100	212	4.1±2.0	74.0±41.6	30.8±24.7	–
LL	78	219	4.5±3.1	68.1±43.6	29.4±24.8	–
LLRR	78	266	2.2±1.3	60.7±42.6	24.4±24.1	–
RR	100	728	2.0±0.1	103.9±36.0	45.8±20.7	–
Ops	–	20	–	20.9±5.5	4.6±1.2	2.4±0.5

8. Related Work

Fine-Morris et al. (2020) uses a similar technique to learn methods with numeric preconditions, leveraging Word2Vec and clustering to learn subgoals, which it uses to partition traces and learn a hierarchy of landmark methods and binary right-recursive methods. The crucial difference between this work and the previous one is that T2N allows for goals to be numeric fluents, and tests the method-learner on a domain with numeric-only subgoals. This enables the learned methods to solve problems where, for example, there is a minimum number of resources needed.

Word2HTN (Gopalakrishnan et al., 2018) uses a technique involving Word2Vec and clustering similar to this work. Unlike Word2HTN, which learns a binary hierarchy (two subtasks per method), T2N learns landmark methods that can decompose a high-level task into as many as $|LM| + 1$ subtasks. Additionally, while Word2HTN can learn numeric preconditions and effects involving arithmetic operations only (e.g., addition, subtraction), T2N learns more complex functions via function composition. This has trade-offs: Word2HTN can simplify its numeric expressions (e.g., by reducing $varX + 3$ and $varX - 2$ into $varX + 1$) in ways T2N cannot.

Both HTN-Maker (Hogg et al., 2016) and HTNLearn (Zhuo et al., 2014) learn task decomposition methods from traces and annotated task definitions comprised of (precondition, effects) pairs. HTNLearn uses constraint satisfaction to learn method preconditions, while HTN-Maker uses goal regression to learn right-recursive subtasks. Crucially, T2N is not given the subtasks as input; while we annotate our traces with the goal task, we identify the bounds of a task using the occurrence of learned landmarks in the effects of an action and do not require subtask specifications. Also, neither handles numeric fluents. Additionally, as our RR library of methods is of comparable structure to the HTNs that HTN-Maker learns, our results show that method libraries with landmark methods (CL, LL, and LLRR) solve problems more quickly than the HTN-Maker-style library (RR).

Segura-Muros et al. (2015) learns HTN planning domains from plan traces using a combination of process mining and inductive learning. Process mining builds a behavioral model from a set of event logs. By treating plan traces as event logs the authors use process mining to learn the hierarchical structure of the plan traces. Once the structure is learned, they extract pre-state and post-state pairs from the plan traces for each action and method and utilize inductive learning to generate preconditions and effects. They show that their algorithm can learn a simple and straightforward domain capable of consistently solving test problems. However, they do not handle numerics.

ICARUS (Choi & Langley, 2005) uses background knowledge and means-ends analysis to learn Teleoreactive logic programs from provided plans. The background knowledge includes concept definitions, which are composed together to form more complex concepts. This hierarchy of concepts defines the hierarchical structure of the programs. Since tasks are linked to the achievement of concepts that are built upon goal-achievement, Teleoreactive logic programs basically encode HGNs. T2N requires less user-provided domain information on hierarchical structure, instead inferring structure from the learned landmarks.

X-Learn (Reddy & Tadepalli, 1997) learns purely-symbolic d-rules with preconditions and a sequence of fully-ground subgoals (similar to HGN decomposition methods or macro-actions) from increasingly-difficult exercises via inductive learning. By contrast, T2N learns numeric goals.

9. Final Remarks

We have presented modifications to an automated learner that allows it to learn hierarchical methods in domains where transitions between subproblems are marked by changes in numeric values. We have discussed our results from learning several libraries of HTN methods in a domain with exclusively-numeric effects using an automated learner, T2N, which we have augmented to allow learning numeric goals. This involved enabling T2N to use numeric conditions as subgoals, where previously it could not, and modifying the precondition-learning procedure to learn context-sensitive numeric conditions, by grounding certain numeric state-variables so that repeatable tasks are attempted only as many times as the training data demonstrates is useful.

We tested the performance of each learned library on a set of test problems and compared their performance, confirming our expectation that libraries structured with domain landmarks are more efficient than libraries structured in HTN-Maker-style with exclusively binary right recursive subtasks. We also found that, while less intuitive than the hand-selected custom landmarks (CL), the learned landmarks (LL, LLRR) resulted in more-efficient methods in terms of planning time, and provided no significant decrease in plan length and only minimal reduction in the amount of backtracking. The only real advantage of the CL set was that it covered all tasks, but modifications to the landmarks selection procedure could remove even this advantage.

The most promising areas for future work are (1) refining domain-landmark selection to improve goal coverage, (2) learning fewer redundant methods or eliminating them in post-processing, (3) optimizing the preconditions to reduce their size, and (4) investigating other ways to decompose the subtraces between each landmark, as both binary right-recursive and flat decomposition have drawbacks (the first makes the method library more computationally expensive, the second makes the method library less flexible). There are two possible ways to resolve (1). The first is to change trace linearization for landmark learning to include the full states (instead of limiting the conditions to the subset in the preconditions and effects of the preceding and following actions). This would create more contextual links between words which might improve landmark selection. The second is to enforce that selected landmarks must occur in at least one trace for each final task. We could accomplish (2) by attempting to find methods with the same (task, subtask) pairs and identifying which has preconditions that subsume the others. Alternatively, we could attempt to solve each task and landmark task to confirm the need for a new method before learning it. Accomplishing (3) is difficult because the precondition calculations are black boxes to the precondition learner. A post-processing step could use domain knowledge and inductive learning to simplify the calculations. (4) could be done by grouping subtraces by subtask and reapplying the landmark selection procedure on each subtrace-group. This would create multiple levels of (task, landmark-tasks) decomposition, possibly allowing for shallower decompositions trees.

Acknowledgements

We thank NRL and ONR for funding this research and NSF's Independent Research and Development (IR/D) Plan. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

Bundy, A., & Wallen, L. (1984). Skolemization. In A. Bundy & L. Wallen (Eds.), *Catalogue of Artificial Intelligence Tools*, 123–123. Berlin, Heidelberg: Springer.

Choi, D., & Langley, P. (2005). Learning Teleoreactive Logic Programs from Problem Solving. *Inductive Logic Programming* (pp. 51–68). Berlin, Heidelberg: Springer.

Fine-Morris, M., Auslander, B., Floyd, M. W., Pennisi, G., Muñoz-Avila, H., & Gupta, K. M. (2020). Learning Hierarchical Task Networks with Landmarks and Numeric Fluents by Combining Symbolic and Numeric Regression. *Proc. of the 8th Annual Conf. on Advances in Cognitive Systems* (p. 16).

Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated Planning: Theory and Practice*. Elsevier.

Gopalakrishnan, S., Munoz-Avila, H., & Kuter, U. (2018). Learning Task Hierarchies Using Statistical Semantics and Goal Reasoning. *AI Communications*, *31*, 167–180.

Hoffmann, J., Porteous, J., & Sebastia, L. (2004). Ordered Landmarks in Planning. *Journal of Artificial Intelligence Research*, *22*, 215–278.

Hogg, C., Muñoz-Avila, H., & Kuter, U. (2016). Learning Hierarchical Task Models from Input Traces. *Computational Intelligence*, *32*, 3–48.

Langley, P., Choi, D., & Rogers, S. (2007). Interleaving Learning, Problem Solving, and Execution in the Icarus Architecture. (p. 27).

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. *Advances in Neural Information Processing Systems*. Curran Associates, Inc.

Nau, D., Muñoz-Avila, H., Cao, Y., Lotem, A., & Mitchell, S. (2001). Total-Order Planning with Partially Ordered Subtasks. *Proc. of the 17th international joint Conf. on Artificial intelligence - Volume 1* (pp. 425–430). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Porteous, J., Sebastia, L., & Hoffmann, J. (2014). On the Extraction, Ordering, and Usage of Landmarks in Planning. *Sixth European Conf. on Planning*.

Reddy, C., & Tadepalli, P. (1997). Learning Goal-Decomposition Rules using Exercises. *Proc. of the fourteenth national Conf. on artificial intelligence and ninth Conf. on Innovative applications of artificial intelligence.* (pp. 843–843).

Segura-Muros, J. A., Pérez, R., & Fernández-Olivares, J. (2015). Learning HTN Domains using Process Mining and Data Mining techniques. *Workshop on Generalized Planning (ICAPS-17)* (p. 8).

Shivashankar, V., Kuter, U., Nau, D., & Alford, R. (2012). A Hierarchical Goal-Based Formalism and Algorithm for Single-Agent Planning. *Proc. of the 11th International Conf. on Autonomous Agents and Multiagent Systems* (p. 9). Valencia, Spain.

Zhuo, H. H., Muñoz-Avila, H., & Yang, Q. (2014). Learning hierarchical task network domains from partially observed plan traces. *Artificial Intelligence*, *212*, 134–157.