Learning Hierarchical Task Networks with Landmarks and Numeric Fluents by Combining Symbolic and Numeric Regression

Morgan Fine-Morris[1] MOF217@LEHIGH.EDU
Bryan Auslander[2] BRYAN.AUSLANDER@KNEXUSRESEARCH.COM
Michael W. Floyd[2] MICHAEL.FLOYD@KNEXUSRESEARCH.COM
Greg Pennisi[2] GREG.PENNISI@KNEXUSRESEARCH.COM
Héctor Muñoz-Avila[1] HEM4@LEHIGH.EDU
Kalyan Moy Gupta[2] KALYAN.GUPTA@KNEXUSRESEARCH.COM
[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

Abstract

In this paper we present an algorithm for the automated learning of hierarchical task networks. Our algorithm has three characteristics: (1) it learns the high-level structure of the domain as landmarks, conditions that always hold in any solution of the problem. (2) It learns right recursive HTNs to fill knowledge gaps between the landmarks. (3) It simultaneously learns symbolic preconditions, such as the need for a path connecting locations, and numeric preconditions such as the minimum amount of fuel needed to complete the path.

1. Introduction

Humans learn complex skills by recursively acquiring and combining simpler skills (Choi & Langley, 2005). Many domains are amenable to hierarchical problem-solving representations whereby complex problems are represented and solved at different levels of abstraction. Examples include (1) some navigation tasks where hierarchical A* has been shown to be a natural solution solving the navigation problem over different levels of abstraction (Holte et al., 1995; Wang et al., 2014); (2) dividing a reinforcement learning task into subtasks where policy control is learned for subproblems and combined to form a solution for the overall problem (Dayan & Hinton, 1993; Dietterich, 2000; Diuk et al., 2013); (3) abstraction planning, where concrete problems are transformed into abstract problem formulations, these abstract problems are solved as abstract plans, and in turn these abstract plans are refined into concrete solutions (Knoblock, 1994; Bergmann & Wilke, 1995); (4) hierarchical task network (HTN) planning where complex tasks are recursively decomposed into simpler tasks (Currie & Tate, 1991; Wilkins, 1999; Erol et al., 1994; Nau et al., 1999); and (5) some cognitive architectures use hierarchical knowledge representations (Choi & Langley, 2005; Laird et al., 1987). These paradigms have in common a divide-and-conquer method to problem solving that is amenable to a stratified representation of the subproblems. Among the various formalisms, HTN planning has been a recurrent research focus over the years. An HTN planner formulates a plan using actions and HTN methods. The latter describe how and when to reduce complex tasks into simpler subtasks. HTN methods are used to recursively decompose tasks until so-called primitive tasks are reached corresponding to actions that can be performed directly in the world. HTNs provide a natural knowledge-modeling representation for many domains Nau et al. (2005), including military planning Mitchell (1997); Muñoz-Avila et al. (1999), strategy formulation in computer games (Hoang et al., 2005; Gorniak & Davis, 2007), manufacturing processes (Nau, 1994; Tao et al., 2008), project planning (Tate, 1976; Ullrich, 2005), story-telling (Cavazza et al., 2002), web service composition (Kuter et al., 2005), and UAV planning Gancet et al. (2005).

Despite these successes, HTN planning can be difficult to use effectively as it requires that a domain expert hand-formulate the methods that decompose the tasks. Automation of method creation from example traces can reduce or obviate the need for a domain expert.

In this paper we report on our work on learning HTN methods automatically given (1) an action model, describing actions as *(name, preconditions, effects)* triples; (2) a collection of action traces annotated with their partial intermediate states; and (3) a collection of tasks. We make a three-fold contribution to the state-of-the-art:

- **We learn methods capturing high-level domain structure, following the work of Gopalakrishnan et al. (2016)**. Specifically, they capture planning landmarks, conditions or events that are true when solving a particular class of problems. For example, when traveling from Jersey City to Manhattan by road, most trips will either cross the Lincoln or Holland tunnels or the George Washington bridge. We call these methods landmark methods. Unlike the ones learned in Gopalakrishnan et al. (2016), our learned landmark methods can consist of arbitrary-many subtasks and not just two.

- **We learn right-recursive methods, following the work of Hogg et al. (2008)**. These methods have the structure *(t P (t' t))*, where t is a compound task, P are the preconditions and t' is a primitive task. As explained in Hogg et al. (2009), right recursive methods "advance" the state by applying t' and then try to solve the same task t again from the new state. Their main drawback is that they don't capture the high-level structure of the domain. In our work, the landmark methods accomplish this. We use right-recursive methods to fill in the gaps of the decomposition where no meaningful landmarks occur.

- **We learn and combine numerical conditions.** In our work, learned methods contain preconditions about numeric conditions such as $res > 0$, where res is the projected remaining fuel level of a vehicle when reaching its destination. To learn these preconditions we extend classical goal regression Reiter (1991) to regress numeric functions.

2. Related Work

Word2HTN (Gopalakrishnan et al., 2016) learns a task hierarchy and a set of methods from plan traces using the natural language processing algorithm Word2vec (Mikolov et al., 2013). Word2vec learns vector representations for words in a corpus based on their context, i.e., their proximity to other words in the corpus. Word2HTN learns the vector representations of plan elements, and by clustering vectors, determines which plan elements have the same contexts and which plan elements occur between separate contexts: those that appear between separate clusters. Methods produced by Word2HTN can only have two subtasks, while our landmark methods can have several subtasks; our algorithm splits problems into several smaller subproblems using a single landmark method, then handles the subtasks right-recursively. While Word2HTN only deals with non-numeric preconditions, our algorithm handles both numeric and non-numeric preconditions.

The problem of learning hierarchical planning knowledge has been a frequent research subject over the years. ICARUS (Choi & Langley, 2005) learns HTN methods by using skills (i.e., abstract definitions of semantics of complex actions) represented as Horn clauses. The crucial step is a teleoreactive process where classical planning is used to fill gaps in the HTN planning knowledge. For example, if the learned HTN knowledge is able to get a package from a starting location to a location $L1$ and the HTN knowledge is also able to get the package from a location $L2$ to its destination, but there is no HTN knowledge on how to get the package from $L1$ to $L2$, then a classical planner is used to generate a plan to get the package from $L1$ to $L2$. That generated plan can then be used to learn new HTN methods to get from $L1$ to $L2$. Another example is HTN-Maker (Hogg et al., 2008). HTN-Maker uses task semantics defined as *(preconditions, effects)* pairs, similar to other works adding task semantics such as TMKs (Murdock, 2001), to identify sequences of contiguous actions in the input plan trace where the preconditions and effects are met. Task hierarchies are learned when an action sequence is identified as achieving a task and the action sequence is a sub-sequence of another larger action sequence achieving another task. This includes the special case when the sub-sequence and the sequence achieve the same task. In this situation, HTN-Maker learns a right recursive task hierarchy. HTN-Maker learns incrementally after each training case is given. Another example is HTNLearn (Zhuo et al., 2014) which transforms the input traces into a constraint satisfaction problem. Like HTN-Maker, it also assumes *(preconditions, effects)* as the task semantics to be given as input. HTNLearn process the input traces converting them into constraints. For example, if a literal p is observed before an action a and a is a candidate first sub-task for a method m, then a constraint c is added indicating that *p is a precondition of m*. These constrains are solved by a MAXSAT solver, which returns the truth value for each constraint. For example, if c is true then p is added as a precondition of m. As a result of the MAXSAT process, HTNLearn is not able to converge to a 100% correct domain (the evaluation of HTNLearn computes the error rates in the learned domain). Similarly, we provide the task definitions for the same purpose: indicating which skills we want to learn. There are two main differences between these works and our work. First, in our work we are learning landmark methods that capture the structure of the domain. Second, we are dealing with domains with numeric values and not just symbolic conditions.

Similar to HTN planning, hierarchical decompositions have been used in *hierarchical reinforcement learning* (Parr & Russell, 1998; Dietterich, 2000). The hierarchical structure of the reinforcement learning problem is analogous to an instance of the decomposition tree that an HTN planner might generate. Given this hierarchical structure, hierarchical reinforcement learners perform value-function composition for a task based on the value functions learned over its subtasks recursively. However, the possible hierarchical decompositions must be provided in advance. In our work, we are learning the task hierarchies.

Hierarchical goal networks (HGNs) (Shivashankar et al., 2012) are an alternative representation formalism to HTNs. In HGNs, goals, instead of tasks, are decomposed at every level of the hierarchy. HGN methods have the same form as HTN methods but instead of decomposing a task, they decompose a goal; analogously instead of subtasks, HGN methods have subgoals. If the domain description is incomplete, HGNs can fall back to STRIPS planners to fill gaps in the domain. On the other hand, total-order HGNs are as expressive as total-order HTNs (Shivashankar, 2015) and its partial-order variant (Shivashankar et al., 2016) is as expressive as partial-order HTNs (Alford et al., 2016). In our work, we define tasks as goals thereby mimicking HGNs although we use the same semantics as general HTNs and in fact we use an HTN planner to solve problems with the learned methods.

Inductive learning has been used to learn rules indicating goal-subgoal relations in X-learn (Reddy & Tadepalli, 1997). This is akin to learning macro-operators (Mooney, 1988; Botea et al., 2005); the learned rules and macro-operators provide search control knowledge to reach the goals more rapidly but they don't add expressibility to standard planning. SOAR learns goal-subgoal relations (Könik & Laird, 2006). It uses as input annotated behavior trace structures, indicating the decisions that led to the plans; this is used to generate goal-subgoal relations. Another work on learning goal-subgoal relations is reported in Ontanón et al. (2010). It uses case-based learning techniques to store goal-subgoal relations, which are then reused by using similarity metrics. These works assume some form of the input traces, unstructured in Ontanón et al. (2010) and structured in Könik & Laird (2006), to be annotated with the subgoals as they are accomplished in the traces. Because in our work, tasks are defined as goals, our learned methods also capture goal-subgoal relations although we use an HTN planning so we don't do "subgoaling" in the STRIPS planning sense.

Goal regression techniques have been used to generate a plan starting from the goals that must be achieved (Reiter, 1991; Pollock, 1998; McDermott, 2002). The result of goal regression can be seen as a hierarchy recursively generated by indicating for each goal what subgoals must be achieved. The goal-subgoal relations resulting from goal regression are a direct consequence of the domain's operators: the goals are effects of the operators and the preconditions are the subgoals. In our work, we extend goal regression to consider numeric functions.

Finally, as we will see in the next the next section, our proposed work is related to the notion of planning landmarks (Hoffmann et al., 2004). Given a **planning problem** P, defined as a triple $(s_0, g, \mathcal{A})$, indicating the initial state s_0, the goals g and the actions $\mathcal{A}$, a planning landmark is either an action $a \in \mathcal{A}$, or state atom $p \in s$ (s is an state, represented as a collection of atoms) that occurs in any solution plan trace solving P. Given the problem description P, planning systems can automatically identify landmarks for P. Planning landmarks have been widely used for automated planning resulting in planners such as LAMA (Richter & Westphal, 2010) and the HGN planner GoDel (Shivashankar et al., 2013).

3. Overview of Learning Systems

The learning system we describe outputs a set of HTN methods H, where each method in H is a 3-tuple *(name preconditions subtasks)*. These methods are learned based on three inputs: a set of traces $T = t_0, ..., t_n$, a set of action definitions A, and a set of task definitions (goals) G.

Each input trace $t_i \in T$ is a sequence of action names and partial states that were observed from an agent operating in the environment:

$$t_i = s_0 \; a_1 \; s_1 \ldots s_{m_i - 1} \; a_{m_i} \; s_{m_i}$$

where m_i is the number of actions performed by the actor in trace t_i (and m_i may be different for each trace in T). Each state s_i is assumed to satisfy the preconditions of a_{i+1} (otherwise the agent would have been unable to perform it) and contain the effects of a_i (i.e., no failed actions). Each action in A is defined as an *(name preconditions effects)* tuple. Tasks in G are associated with specific goals. For example, a task *position(veh23,loc46)* indicates that the vehicle $veh23$ has to reach location $loc46$. This requirement mimics hierarchical goal networks. But we are using a general HTN planner and therefore, in principle, the requirement of tasks as goals could be relaxed by, for example, associating a task with a logical formula to determine if it is valid or not.

The learned methods in H can be split into three categories: *landmark methods*, *right-recursive methods*, and *single-primitive task methods*. Landmark methods have subtasks to reach the landmarks and to reach the final goal. Right-recursive methods have 2 subtasks: a left subtask which is always primitive, and a right subtask which is non-primitive. The single-primitive task methods, as the name implies, contain a single primitive task and terminates the recursion of the right-recursive methods.

4. Learning Steps

Algorithm 1 provides an overview of the system. The main steps include: landmark identification (line 1, Section 4.1), identifying the trace goals and necessary subtasks (line 3, Section 4.2, partitioning the traces based on subtasks (line 4, Section 4.3), right-recursive method learning and forming preconditions (lines 5 and 6, Section 4.4), and landmark method creation (line 7, Section 4.5). We will discuss each of these steps subsequently, in greater detail.

4.1 Step 1: Identifying Common Landmarks

The first step of method learning is landmark identification. This is done by providing a set of traces to the Word2Vec algorithm (Mikolov et al., 2013) to generate an embedding for each atom in the trace. Possible atoms are the preconditions (e.g., *position(veh0,loc1)*), effects (e.g., *position(veh0,loc2)*) and the actions' names (e.g., *transit(veh0,loc1,loc2)*). Each of these atoms is treated as a word and traces are treated as sentences, i.e. for a section of subtrace $a_1 \; s_1 \; a_2$ where s_1 is a state comprised of conditions $c_1, c_2, ..., c_n$, the trace can be written $a_1 \; c_1 \; c_2$ ... $c_n \; a_2$. Note that the condition atoms representing the state can occur in any order between a_1 and a_2. Therefore, state s_1 can be written as $c_1 \; c_2$ ... c_n or equivalently as $c_2 \; c_1$ ... c_n, etc. In our system, the condition atoms are ordered arbitrarily. The context of each atom ϕ is defined by other atoms occurring before and

Algorithm 1 Overview of Learning System

```
Input: A set of traces ($T = t_0, ..., t_n$), a set of operators ($A$), and a set of tasks ($G$).
1: Identify a set of common landmarks, $L$, which are a subset of $G$.
2: for $t_i \in T$ do
3:     Determine the final goal of $t_i$, $g$, and create a new list, $S = L \cdot (g)$.
4:     Split $t_i$ into subsection terminating at first occurrence of each landmark
5:     Build a set of right-recursive methods for each section and add them to $H$, return the pre-
       conditions of highest-level method, grounded appropriately.
6:     Regress over preconditions to form a set of preconditions, $P$
7:     Build a landmark method, $LM = (g, P, S)$.
8:     add $LM$ to $H$
Output: The HTN methods, $H$
```

after ϕ in the traces. Using this context, Word2Vec generates vector representations for these atoms. The resulting vectors are clustered into two groups and we calculate the cosine distance,

$$CosineDist = 1 - \left(\vec{v1} \cdot \vec{v2} / \|v1\| \, \|v2\|\right)$$

between the vector representations of each atom and those of the atoms in the opposite cluster, and take the average. We order the atoms by their average cosine distance, and filter them to remove all but the goal atoms (i.e., atoms occurring in G). From the remaining atoms, those with an average distance lower than a threshold λ are added to the landmark list L. Each landmark list L will contain k_i landmarks, where k_i is the number of landmarks identified in trace t_i (i.e., $l_1, l_2, \ldots l_{k_i}$).

For example, in the transit domain we used in the experiments, when there are bottleneck locations, these are identified as landmarks and they are included in L in the same order they occur in the traces.

4.2 Step 2: Determining the Final Goal of Each Trace

We assume that, for each trace, the observed agent successfully achieved their final goal g in the final action of the trace (i.e., no unnecessary extra actions were performed). To determine the final goal, we take the list of atoms in the final state of the trace (i.e., s_m) and filter out rigid relations (i.e., atoms appearing in all states of the trace) and atoms not in G. Thus, g contains all remaining atoms. In the sample transit domain we use in our examples, the only atom remaining in the final goal is the final location of the vehicle (e.g., *position(veh0, loc10)*. At this point, the algorithm has identified landmarks L and final goals g. These two lists are then combined into a list of subtasks S, terminated by final goals g. This represents that it is necessary to complete subtasks to reach each landmark before finally completing a subtask to reach the goal.

4.3 Step 3: Splitting the Traces

The list of subtasks S for a trace is ordered based on the occurrence of landmarks and final goals in the trace (i.e., $l_1, l_2, \ldots l_{k_i}, g$). For the given trace $t_i = s_0 \; a_1 \; s_1 \ldots s_{m_i-1} \; a_{m_i} \; s_{m_i}$, we split it into

$|S|$ subtraces. Each of those subtraces terminates in either a landmark or a final goal. For example, $s_0\ a_1\ \ldots s_{j_1}$, where l_1 occurs in s_{j_1}; $s_{j_1}\ \ldots s_{j_2}$, where l_2 occurs in s_{j_2}, and so forth until the final trace $s_{j_{k_i}}\ \ldots s_{m_i}$, where g occurs in s_{m_i}.

4.4 Step 4: Learning Right-Recursive Methods

As we presented previously, methods are represented as 3-tuples *(name preconditions subtasks)*. As described in the previous section, each trace is split into subtraces that achieve a specific subtask (i.e., a landmark or final goal). Each of these subtraces are processed to create a hierarchy of right-recursive methods.

Consider the subtrace $\pi_{i:j} = s_i\ a_{i+1}\ s_{i+1} \ldots s_{j-1}\ a_j\ s_j$ that achieves subtask g_j. Our approach constructs a right-recursive method $(g_j, P_{i+1}, (a_{i+1}\ g_j))$, where it produces a name representing the subtask it is achieving (i.e., g_j), method preconditions[1] P_{i+1} calculated via goal regression over the actions $a_{i+1}...a_j$, and a subtask list (i.e., $(a_{i+1}\ g_j)$). The subtask list starts with a single-primitive task, performing a_{i+1}, followed by another task to continue attempting to achieve g_j. This process is performed iteratively, with each iteration shortening the subtrace by removing the first state and action in the subtrace (e.g., s_i and a_{1+1}) and generating a new right-recursive method on the subtrace (e.g., $\pi_{(i+1):j} = s_{i+1} \ldots s_{j-1}\ a_j\ s_j$). The process terminates when only a single action a_j remains in the subtrace and a method with only a single subtask is created: (g_j, P_{a_j}, a_j). The result of this process is $j - i + 1$ methods, all of which have the same name (i.e., g_j). Thus, any right-recursive method with g_j in it's subtask list can use any of of the methods with that name for which the preconditions hold. This is important as it does not require actions to be performed only in the order they appeared in the subtrace used to learn the right-recursive methods. All methods that are created using this recursive approach are added to the method set H, except when identical methods already exist.

To learn the preconditions (e.g., P_{i+1}) for each of the methods, we regress over the subtrace $\pi_{i:j}$. We extend classical goal regression (Reiter, 1991) to account for numeric conditions that occur in the traces as follows. We find the set $P_{i+1} = \mathcal{R}(g_j, \pi_{i:j})$ with the regression operator $\mathcal{R}$ on an trace $\pi_{i:j}$. The value of $\mathcal{R}(g_j, \pi_{i:j})$ is the minimal set of atoms that must be true in the state s_i to guarantee that $a_{i+1} \ldots a_j$ can be executed and g_j is achieved:

- If $\pi_{i:j}$ is the empty action trace, then $\mathcal{R}(g_j, \pi_{i:j}) = g_j$.
- If $\pi_{i:j}$ contains a single action $a = (name, P_a, E_a)$, then $\mathcal{R}(g_j, \pi_{i:j}) = g_j \ominus P_a$. If (v, α) is a (variable, value) pair in P_a, then we assign α to v in g_j. For all other variables in g_j, their values remain the same (the operator $\ominus$ is defined in Reifsnyder & Munoz-Avila (2020)).
- If $\pi_{i:j}$ contains two or more actions $a_{i+1} \ldots a_j$, then $\mathcal{R}(g_j, \pi_{i:j}) = \mathcal{R}(\mathcal{R}(g_j, a_j), a_{i+1}a_{i+2} \ldots a_{j-1})$.

Given a trace containing the actions $a_{i+1}\ \ldots\ a_j$ achieving a goal g_j on a variable v, we first regress over $a_{j-1}\ a_j$. Suppose that the preconditions $P_{a_{j-1}}$ of a_{j-1} uses a function f_{j-1} which accepts variable v as an argument and stores its result $f_{j-1}(v)$ to temporary variable res_{j-1}. The

1. To clarify, P_i represents the preconditions of the ith right-recursive method, which we learn, and P_{a_i} the preconditions of the ith action.

task	position(?veh0, ?loc0)
preconditions	position(?veh0, ?loc1)
	gas_tank_level(?veh0, ?gtl)
	...
	$?res0 \leftarrow \lvert ?xloc0 - ?xloc1 \rvert + \lvert ?yloc0 - ?yloc1 \rvert$
	$?res1 \leftarrow ?gtl - \frac{?res0}{60}$
	$?res1 > 0$
subtasks	TransitAction(?veh0, ?loc1, ?loc0)
task	position(?veh0, ?loc0)
preconditions	position(?veh0, ?loc1)
	gas_tank_level(?veh0, ?gtl)
	...
	$?res0 \leftarrow \lvert ?xloc2 - ?xloc1 \rvert + \lvert ?yloc2 - ?yloc1 \rvert$
	$?res1 \leftarrow ?gtl - \frac{?res0}{60}$
	$?res1 > 0$
	$?res2 \leftarrow \lvert ?xloc2 - ?xloc0 \rvert + \lvert ?yloc2 - ?yloc0 \rvert$
	$?res3 \leftarrow ?res1 - \frac{?res2}{60}$
	$?res3 > 0$
subtasks	TransitAction(?veh0, ?loc1, ?loc2)
	position(?veh0, ?loc0)

Table 1. A primitive method and a right recursive method

effects $E_{a_{j-1}}$ of the action a_{j-1} may then use res_{j-1} to update v. Action a_j may have preconditions P_{a_j} that use a function f_j which accepts v as input. Regressing over a_{j-1} a_j results in a new set of preconditions which directs the output of f_{j-1} to the input of f_j such that the combined preconditions will calculate $f_j(f_{j-1}(v))$ or, equivalently $res_{j-1} \leftarrow f_{j-1}(v); f_j(res_{j-1})$. We continue regressing numerical conditions on the rest of the trace in the same way.

Example. Table 1 shows an example of two methods decomposing the task *position(?veh0,?loc0)* (the question mark prefix of the name, indicates a variable). The first decomposes it into a single primitive subtask *TransitAction(?veh0, ?loc1, ?loc0)*. The second is a right recursive method that decomposes it into the recursion *(TransitAction(?veh0, ?loc1, ?loc2) position(?veh0, ?loc0))*. The preconditions of the first method match those of *TransitAction* (for the sake of space only a subset of the preconditions are shown). The preconditions of the first method in Table 1 require that $?veh0$ has an initial position $?loc1$, and an initial gas tank level, $?gtl$. The final 3 preconditions calculate the distance between $?loc0$ and $?loc1$, from their coordinates ($?xloc0$, $?yloc0$) and ($?xloc1$, $?yloc1$), stores the result in a temporary variable $?res0$, and calculates the projected value for $?gtl$ as a function of the initial value of $?gtl$ and the distance in $?res0$. The method requires that the projected value for $?gtl$ be greater than or equal to zero. The recursive method in Table 1 decomposes the task $position(?veh0, ?loc1)$ where the initial locations, $?loc0$ and the final location $?loc1$ are separated

task	position($?veh0, ?loc0$)
preconditions	position($?veh0, ?loc1$)
	gas_tank_level($?veh0, ?gtl$)
	...
	$?res0 \leftarrow \lvert ?xloc0 - ?xloc2 \rvert + \lvert ?yloc0 - ?yloc2 \rvert$
	$?res1 \leftarrow ?gtl - \frac{?res0}{60}$
	$?res1 > 0$
	...
	$?res15 \leftarrow \lvert ?xlm2 - ?xloc1 \rvert + \lvert ?ylm2 - ?yloc1 \rvert$
	$?res16 \leftarrow ?res14 - \frac{?res15}{60}$
	$?res16 > 0$
subtasks	position($?v1, lm0$)
	position($?veh0, lm1$)
	position($?veh0, lm2$)
	position($?veh0, ?loc0$)

Table 2. Example Learned Landmark Method.

by an intervening location, $?loc2$. The preconditions are similar to to the first method, but calculates two distances, from coordinates $(?xloc0, ?yloc0)$ to $(?xloc2, ?yloc2)$ to $(?xloc1, ?yloc1)$. These are the coordinates of locations $?loc0$, $?loc2$, and $?loc1$ respectively. It then calculates the expected value of $?gtl$ after traveling from $?loc0$ to $?loc2$, and checks that the gas in $?veh0$ has not been depleted (i.e., that $?gtl > 0$). If not, it calculates the expected value of $?gtl$ after traveling from $?loc2$ to $?loc1$, and checks that the gas has not been depleted.

4.5 Step 6: Building Landmark Methods

The right-recursive methods described previously, while valuable, suffer in that they have at most two subtasks (i.e., a primitive action and another right-recursive subtask). The landmark methods allow for more potential subtasks and the grounding of some elements (e.g., landmark locations). At this point, a given input trace has (1) its landmarks identified ($l_1, l_2, \ldots, l_{k_i}$), (2) its goal identified (g), (3) a combined subtask list of landmarks and goals (S), (4) a set of right-recursive methods capable of achieving the task accomplished by each landmark-or-goal-terminated subtrace, and (5) a set of preconditions (P) formed by performing goal regression over the entire trace. This learned information can now be combined into a more complex landmark method. Thus, a landmark method $LM = (g, P, S)$ is created for each trace and added to the set of methods H. At this stage, H contains both right-recursive methods and landmark methods, and landmark methods use right-recursive methods to perform subtasks. It should be noted that when using these planning methods, a planner is able to use either the more complex landmark methods or the right-recursive methods, depending on the planning problem.

Example. Table 2 show a landmark method with 4 subtasks. Several preconditions are omitted to reduce length. The landmarks locations are $lm0, lm1, lm2$, and are constants representing fixed locations. As with the methods in Table 1, the method requires that vehicle $?veh0$ have an initial position $?loc1$ and an initial gas tank level of $?gtl$. The first 3 equations show the calculation of the distance between the first locations in the sequence, and the resulting change in $?gtl$, while that last set of equations shows the distance calculation between the last landmark locations $lm2$ and the final destination location $?loc1$. Not shown are the distance calculations for the location pairs visited between $?loc2$ and $lm2$ and the resulting changes to $?gtl$, which dictate the value of $?res14$.

5. Empirical Evaluation

5.1 Experimental setup

We generated a set of training traces from a map which we used to learn two sets of methods, the landmark set and the right-recursive set. The landmark set consists of the landmark methods, the right-recursive methods learned from subtraces between the landmarks and the single-primitive-task methods. The right-recursive set consists of right-recursive methods learned from the whole traces and single-primitive-task methods. We then generate solvable and unsolvable test problems on 3 different maps, described below, and attempt to plan on them.

Training Map. We generated a hierarchical map comprised of 6 sections; 2 central sections and 4 satellite sections. We hard-coded the large-scale structure of the map, and randomly generated a set of nodes and edges for each map section. The design of the map was meant to emulate a geographic map with strict bottlenecks limiting movement between sectors. Each satellite section was connected to one central section, and the central sections were connected to one another.

Training traces. We designated two satellite sections connected to the same center section as the start sections, and the two sections connected to the other center section as the end sections. We allowed the start and destination location for a vehicle to be any location in the start and end sections, respectively. These restrictions ensured that the training traces had consistent landmarks. We constructed an initial set of training traces from which to learn methods by creating one problem for each combination of start and end location, and finding the shortest weighted path between the locations for each problem. Then we filtered them, retaining one problem for every combination of path lengths between the start location and first landmark location, and destination location and last landmark location. This resulted in 20 learning problems.

The traces used by Word2Vec to learn the landmark atoms contained a RefuelAction whereas the traces used to learn the methods after learning the landmark atoms contained no refuel actions. We exclude the refuel action so we can check if the methods correctly learned the fuel level needed to traverse between locations (otherwise every problem would be solvable by simply refueling).

Word2Vec settings. Word2Vec accepts a collection of tokenized documents as input, where each word in the document is a separate token. In our use of Word2Vec, each trace is a document and each atom is a word. Word2Vec requires a large corpus, so we increased the number of training traces by creating copies of each trace and randomly shuffling the order of atoms within each partial state, for a corpus of 10200 traces. We set the initial learning rate alpha as 0.0001, which drops linearly to the minimum alpha of 1e-05 over the course of training. We set the context-window size to 20% of the average trace length, the vector size to approximately $2.5 \times |vocabulary|$, the minimum frequency count to the average of the total frequency for all words, the downsampling frequency to $(1e - 5/10)$ and negative sampling value to 5. We train for 7000 epochs using the skip-gram configuration.

Parameters. The training map had 34 nodes and an average node connectivity of 1.19. The average number of nodes per section was 5.67 ± 0.52 and the average node connectivity per section was 2.45 ± 0.94. Test map 2 had 59 nodes and a total average connectivity of 3.53. The 10 sections had an average of 5.9 ± 0.74 nodes and an average connectivity of 2.33 ± 0.83. Test map 3 had 25 nodes and an average connectivity of 2.90.

Learned Method Sets. The landmark method set includes 17 landmark methods and 4 right-recursive and 1 single-primitive task methods. The right-recursive set contains 12 right-recursive and 1 single-primitive task methods.

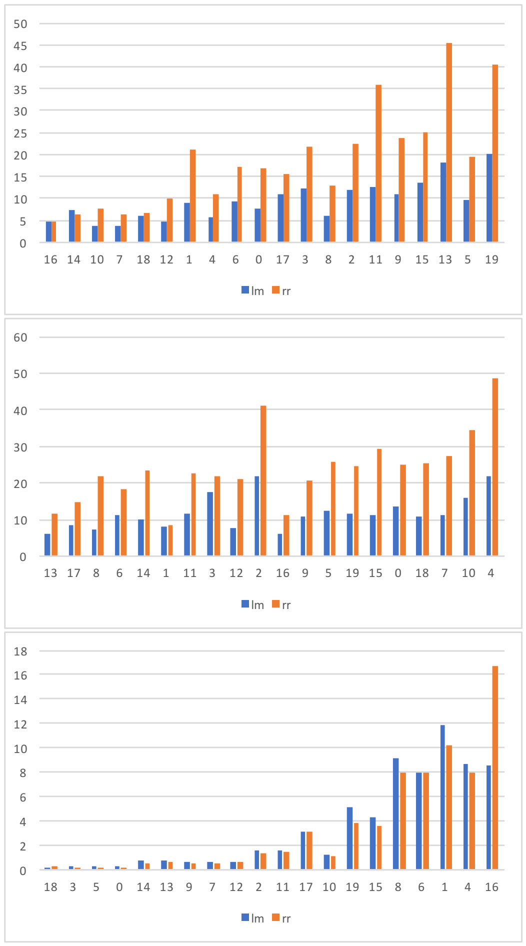
Testing Maps, Problems, and Evaluation. We test our learned method sets on 3 maps. The first is the training map, the second is a modified version of the training map, and the third is a randomly generated map. To create the second map from the training map, we added 2 new satellite sections to each central lobe (for a total of four new satellite sections) so as to ensure that the landmarks are still true global landmarks (i.e., that any path starting at a designated start location on the map would have to pass through each landmark location in the appropriate order to reach the destination). For the third map, the random geographic map, we ensured that for all node pairs the shortest path between them visited no more nodes than the longest path in the training examples. The third map did not contain the landmark locations.

For each test map, we create a random set of 25 problems on the map. We allow vehicles in 20 of the problems to have enough gas to traverse the shortest path to the destination location. We ensure that the vehicles of the remaining 5 do not have enough gas to reach their destination. We then attempt to solve each problem, 3 times each, using the landmark and right-recursive sets. When solving the test problems from map 3, which did not have the landmark locations, we converted the landmark locations into variables in the landmark methods.

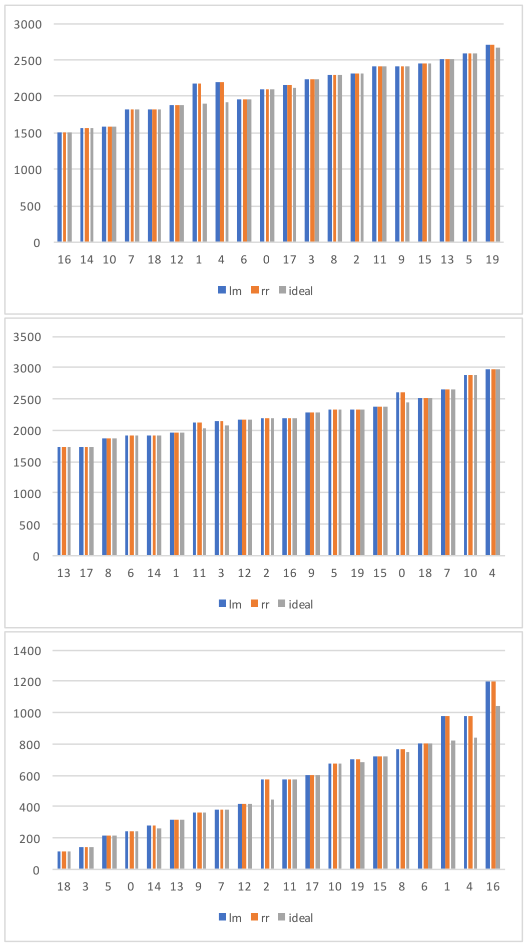
Metrics. For each set of problems and each set of methods we measured three metrics: problems solved, distance traveled, and average planning runtime (average over three planning attempts). We also calculated the travel distance of the shortest solution-path between the locations.

5.2 Results

Both method sets were able to solve all solvable problems, and no method set was able to solve an unsolvable problem. Figure 1 shows the distance traveled and average planning time for all solved problems. The x-axis of each plot shows the problem number, sorted from smallest to largest according to the length of the shortest path. Both method sets found paths of the same length, and both found a path of ideal length on most of the problems. For most problems, the landmark method set has a clear advantage in runtime on maps 1 and 2, in some cases finding a plan in half the time it took the right-recursive method set, although on map 3 no method set was consistently faster than the other.



(a) Average Planning Runtime



(b) Travel Distances

Figure 1. Metrics for the solvable problems. Label lm (blue) refers to the landmark method set. Label rr (orange) refers to the right-recursive method set. The x-axis of each figure indicates problem number, and is sorted according to the travel distance of the vehicle in the ideal solution. The y-axis of figures in column (a) show time in seconds. The y-axis of figures in column (b) show arbitrary units of distance.
From top to bottom, the rows correspond to map 1, 2 and 3.

6. Final Remarks

We have presented an algorithm that accepts plan traces annotated with intermediate partial states, identifies high-level subtasks and then learns HTN methods based on these subtasks, learns right-recursive methods to fill the gaps between those subtasks and combines the numeric preconditions of the actions in the trace to determine the numeric preconditions of the methods. We have tested the performance of the resulting methods on transit domain problems generated for 3 different maps.

Our results demonstrate that for a map with distinct landmarks, landmark methods provide a consistent and noticeable increase in planning speed over right-recursive methods, without a change in plan efficiency (measured by plan travel distance). For maps without landmarks, the trend is less consistent, but landmark methods appear to coincide with a slight decrease in planning speed, but no loss of plan efficiency.

Future work includes incorporating techniques to increase the flexibility and transferability of the learned landmark methods, so that the methods can easily be applied to a new map with new landmarks. Two major components of this are 1) developing a technique to quickly identify landmark locations in new maps, so that landmark methods can be reused on a new map by replacing the grounded landmarks, and 2) learning methods that are agnostic or semi-agnostic to the path between locations so that methods do not require a specific number of edges between the start, destination, and landmark locations. The first component can potentially be accomplished by statistical analysis on maps, to identify transferable characteristics of landmark locations. These characteristics can be used to quickly identify landmarks on a new map, without having to retrain methods from new example traces. The second component may involve inductive learning (Michalski, 1983) to generalize from common characteristics of methods decomposing similar tasks to more general methods for decomposing that task. We will also test our algorithms on larger maps and other domains.

Acknowledgements

This research was supported by ONR under grants N00014-18-1-2009 and N68335-18-C-4027 and NSF grant 1909879.

References

Alford, R., Shivashankar, V., Roberts, M., Frank, J., & Aha, D. W. (2016). Hierarchical planning: Relating task and goal decomposition with task sharing. *International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 3022–3029).

Bergmann, R., & Wilke, W. (1995). Building and refining abstract planning cases by change of representation language. *Journal of Artificial Intelligence Research (JAIR)*, *3*, 53–118.

Botea, A., Müller, M., & Schaeffer, J. (2005). Learning partial-order macros from solutions. *International Conference on Automated Planning and Scheduling (ICAPS)* (pp. 231–240).

Cavazza, M., Charles, F., & Mead, S. J. (2002). Interacting with virtual characters in interactive storytelling. *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1* (pp. 318–325). ACM.

Choi, D., & Langley, P. (2005). Learning teleoreactive logic programs from problem solving. *International Conference on Inductive Logic Programming* (pp. 51–68).

Currie, K., & Tate, A. (1991). O-Plan: The open planning architecture. *Artificial Intelligence*, *52*, 49–86.

Dayan, P., & Hinton, G. E. (1993). Feudal reinforcement learning. *Advances in neural information processing systems* (pp. 271–278).

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research (JAIR)*, *13*, 227–303.

Diuk, C., Schapiro, A., Córdova, N., Ribas-Fernandes, J., Niv, Y., & Botvinick, M. (2013). Divide and conquer: hierarchical reinforcement learning and task decomposition in humans. In *Computational and robotic models of the hierarchical organization of behavior*, 271–291. Springer.

Erol, K., Hendler, J., & Nau, D. S. (1994). HTN planning: Complexity and expressivity. *National Conference on Artificial Intelligence (AAAI)* (pp. 1123–1128).

Gancet, J., Hattenberger, G., Alami, R., & Lacroix, S. (2005). Task planning and control for a multi-uav system: architecture and algorithms. *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on* (pp. 1017–1022). IEEE.

Gopalakrishnan, S., Munoz-Avila, H., & Kuter, U. (2016). Word2htn: Learning task hierarchies using statistical semantics and goal reasoning. *The IJCAI-2016 Workshop on Goal Reasoning*. AAAI.

Gorniak, P., & Davis, I. (2007). Squadsmart: Hierarchical planning and coordinated plan execution for squads of characters. *AIIDE* (pp. 14–19).

Hoang, H., Lee-Urban, S., & Muńoz-Avila, H. (2005). Hierarchical plan representations for encoding strategic game AI. *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*.

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

Hogg, C., Kuter, U., & Muñoz-Avila, H. (2009). Learning hierarchical task networks for nondeterministic planning domains. *Twenty-First International Joint Conference on Artificial Intelligence*.

Hogg, C., Muñoz-Avila, H., & Kuter, U. (2008). HTN-MAKER: Learning HTNs with minimal additional knowledge engineering required. *Conference on Artificial Intelligence (AAAI)* (pp. 950–956). AAAI Press.

Holte, R. C., Perez, M., Zimmer, R., & MacDonald, A. (1995). Hierarchical a*. *Symposium on Abstraction, Reformulation, and Approximation*.

Knoblock, C. A. (1994). Automatically generating abstractions for planning. *Artificial intelligence*, *68*, 243–302.

Könik, T., & Laird, J. E. (2006). Learning goal hierarchies from structured observations and expert annotations. *Machine Learning*, *64*, 263–287.

Kuter, U., Sirin, E., Nau, D. S., Parsia, B., & Hendler, J. (2005). Information gathering during planning for web service composition. *Journal of Web Semantics (JWS)*, *3*, 183–205.

Laird, J., Newell, A., & Rosenbloom, P. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence*, *33*, 1–67.

McDermott, D. V. (2002). Estimated-regression planning for interactions with web services. *AIPS* (pp. 204–211).

Michalski, R. S. (1983). A theory and methodology of inductive learning. In *Machine learning*, 83–134. Springer.

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* (pp. 3111–3119).

Mitchell, S. (1997). A hybrid architecture for real-time mixed-initiative planning and control. *Innovative Applications of Artificial Intelligence Conference (IAAI)* (pp. 1032–1037).

Mooney, R. J. (1988). Generalizing the order of operators in macro-operators. *Machine Learning* (pp. 270–283).

Muñoz-Avila, H., McFarlane, D., Aha, D. W., Ballas, J., Breslow, L., & Nau, D. S. (1999). Using guidelines to constrain interactive case-based HTN planning. *International Conference on Case-Based Reasoning (ICCBR)* (pp. 288–302).

Murdock, J. W. (2001). *Self-improvement through self-understanding: Model-based reflection for agent adaptation*. Doctoral dissertation, Georgia Institute of Technology.

Nau, D. S. (1994). Manufacturing-operation planning vs AI planning. *Third International Conference on Information and Knowledge Management*.

Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Muñoz-Avila, H., Murdock, J. W., Wu, D., & Yaman, F. (2005). Applications of SHOP and SHOP2. *IEEE Intelligent Systems*, *20*, 34–41.

Nau, D. S., Cao, Y., Lotem, A., & Muñoz-Avila, H. (1999). SHOP: Simple hierarchical ordered planner. *International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 968–973). Morgan Kaufmann.

Ontanón, S., Mishra, K., Sugandh, N., & Ram, A. (2010). On-line case-based planning. *Computational Intelligence*, *26*, 84–119.

Parr, R. E., & Russell, S. (1998). *Hierarchical control and learning for markov decision processes*. University of California, Berkeley Berkeley, CA.

Pollock, J. L. (1998). The logical foundations of goal-regression planning in autonomous agents. *Artificial Intelligence*, *106*, 267–334.

Reddy, C., & Tadepalli, P. (1997). Learning goal-decomposition rules using exercises. *International Conference on Machine Learning (ICML)* (pp. 843–851).

Reifsnyder, N., & Munoz-Avila, H. (2020). Policy regression for monitoring execution in goal reasoning systems. *Annual Conference on Advances in Cognitive Systems*.

Reiter, R. (1991). The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz (Ed.), *Artificial intelligence and mathematical theory of computation: Papers in honor of john mccarthy, (ed.)*. Academic Press.

Richter, S., & Westphal, M. (2010). The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, *39*, 127–177.

Shivashankar, V. (2015). *Hierarchical goal network planning: Formalisms and algorithms for planning and acting*. Doctoral dissertation, Dept. of Computer Science, University of Maryland.

Shivashankar, V., Alford, R., Kuter, U., & Nau, D. (2013). The godel planning system: a more perfect union of domain-independent and hierarchical planning. *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence* (pp. 2380–2386). AAAI Press.

Shivashankar, V., Alford, R., Roberts, M., & Aha, D. W. (2016). Cost-optimal algorithms for hierarchical goal network planning: A preliminary report. *ICAPS Workshop on Heuristics and Search for Domain-Independent Planning (HSDIP)*.

Shivashankar, V., Kuter, U., Nau, D., & Alford, R. (2012). A hierarchical goal-based formalism and algorithm for single-agent planning. *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2* (pp. 981–988). International Foundation for Autonomous Agents and Multiagent Systems.

Tao, F., Zhao, D., Hu, Y., & Zhou, Z. (2008). Resource service composition and its optimal-selection based on particle swarm optimization in manufacturing grid system. *IEEE Transactions on industrial informatics*, *4*, 315–327.

Tate, A. (1976). *Project planning using a hierarchic non-linear planner*. Technical Report 25, Department of Artificial Intelligence, University of Edinburgh.

Ullrich, C. (2005). Course generation based on htn planning. *LWA* (pp. 74–79).

Wang, H., Zhou, J., Zheng, G., & Liang, Y. (2014). Has: Hierarchical a-star algorithm for big map navigation in special areas. *Digital Home (ICDH), 2014 5th International Conference on* (pp. 222–225). IEEE.

Wilkins, D. E. (1999). Using the sipe-2 planning system. *Artificial Intelligence Center, SRI International, Menlo Park, CA*.

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