Learning Domain Structure in HGNs for Nondeterministic planning

Morgan Fine-Morris and Héctor Muñoz-Avila
Lehigh University
Bethlehem, PA 18015
{mof217, hem4}@lehigh.edu

Abstract

This paper presents preliminary ideas of our work for automated learning of Hierarchical Goal Networks in nondeterministic domains. We are currently implementing the ideas expressed in this paper.

Introduction

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 and 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 and Wilke 1995); and (4) hierarchical task network (HTN) planning where complex tasks are recursively decomposed into simpler tasks (Currie and Tate 1991; Wilkins 1999; Erol, Hendler, and Nau 1994; Nau et al. 1999). These paradigms have in common a divide-and-conquer method to problem solving that is amenable to 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. The HTN planners SHOP and SHOP2 (Nau et al. 1999; 2003) have routinely demonstrated impressive gains in performance (runtime and otherwise) over standard planners. The primary reason for these performance gains is because of the capability of HTN planners to exploit domain-specific knowledge (Wilkins and desJardins 2001). 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, Lee-Urban, and Muńoz-Avila 2005; Gorniak and Davis 2007), manufacturing processes (Nau 1994; Tao et al. 2008), project planning (Tate 1976; Ullrich 2005), story-telling (Cavazza, Charles, and Mead 2002), web service composition (Kuter et al. 2005), and UAV planning (Gancet et al. 2005)

Despite these successes, HTN planning suffers from a representational flaw centered around the notion of *task*. A task is informally defined as a description of an activity to be performed (e.g., *find the location of robot r15*) (e.g., the task "dislodge red team from Magan hill" in some adversarial game) and syntactically represented as a logical atom (e.g., *(locate r15)*). (e.g., "(dislodge redteam Magan)"). Beyond this syntax, there is no explicit semantics of what tasks actually mean in HTN representations. HTN planners obviate this issue by requiring that a complete collection of tasks and methods is given, one that decomposes every complex task in every plausible situation. However, the knowledge engineering effort of creating a complete set of tasks and methods can be significant (Estlin, Chien, and Wang 1997). Furthermore, researchers have pointed out that the lack of tasks' semantics make using HTNs problematic for execution monitoring problems (Dvorak, Amador, and Starbird 2008; Dvorak et al. 2009). Unlike goals, which are conditions that can be evaluated against the current state of the world, tasks have no explicit semantics other than decomposing them using methods.

For example, suppose that a team of robots is trying to locate r15 and, using HTN planning, it generates a plan calling for the different robots to ascertain r15's location. While executing the plan generate a complex plan in a gaming task to dislodge red team from Magan hill, the HTN planner might set a complex plan to cutoff access to Magan, surround it, weaken the defenders with artillery fire and then proceed to assault it. If sometime while executing the plan, the opponent abandons the hill, the plan would continue to be executed despite the fact that the task is already achieved. This is due to the lack of task semantics, so their fulfillment cannot be checked against the current state; instead their fulfillment is only guaranteed when the execution of the generated plans is completed.

Hierarchical Goal Networks (HGNs) solve these limitations by representing goals (not tasks) at all echelons of the hierarchy (Shivashankar et al. 2012). Hence, goal fulfillment can be directly checked against the current state. In particular, even when a goal g is decomposed into other goals (i.e., in HGN, HGN methods decompose goals into subgoals), the question if the goal is achieved can be answered directly by checking if it is valid in the current state. So in the previous example, when the opponent abandons the hill, an agent executing the plan knows this goal has been achieved regardless of how far it got into executing the said plan.

Another advantage of HGNs is that it relaxes the complete domain requirement of HTN planning (Shivashankar et al. 2013); in HTN planning a complete set of HTN methods for each task is needed to generate plans. Even if the HGN methods are incomplete, it is still possible to generate solution plans by falling back to standard planning techniques such as heuristic planning (Hoffmann and Nebel 2001) to achieve any open goals. Nevertheless, having a collection of well-crafted HGN methods can lead to significant improvement in performance over standard planning techniques (Shivashankar 2015).

When the HGN domain is complete (i.e., there is no need to revert to standard planning techniques to solve any problem in the domain), its expressiveness is equivalent to Simple Hierarchical Ordered Planning (Shivashankar 2015), which is the particular variant of HTN planning used by the widely used SHOP and SHOP2 (Nau et al. 2003) HTN planners. SHOP requires the user to specify a total order of the tasks; SHOP2 drops this requirement allowing partial-order between the tasks (Nau et al. 2001). Both have the same representation capabilities although SHOP2 is usually preferred since it doesn't force the user to provide a total order for the method's subtasks (Nau et al. 2001).

In this work, we propose the automated learning of HGNs for ND domains extending our previous work on learning HTNs for deterministic domains (Gopalakrishnan, Muñoz-Avila, and Kuter 2018). While work exists on learning goal hierarchies (Reddy and Tadepalli 1997; Könik and Laird 2006; Ontanón et al. 2010), these works are based on formalisms that have more limited representations than HGNs and in fact predate them.

Related Work

Aside from HGNs, researchers have explored other ways to address the limitation associated with the lack of tasks' semantics. For instance, TMKs (Task-Method-Knowledge models) require not only the tasks and methods to be given but also the semantics of the tasks themselves as *(preconditions,effects)* pairs (Murdock and Goel 2001; Murdock 2001). While this solves the issue with the lack of tasks' semantics it may exacerbate the knowledge engineering requirement of HTNs: the knowledge engineer must not only encode the methods and tasks but also must encode their semantics and ensure that the methods are consistent with the given tasks' semantics. To deal with incomplete HTN domains, researchers have proposed translating the methods into a collection of actions so that standard planning techniques can be used (Alford, Kuter, and Nau 2009). There are two limitations with this approach. First, HTN planning is strictly more expressive than standard planning (Erol, Hendler, and Nau 1994), hence the translation will be incomplete in many domains. Second, for domains when translating methods into actions is possible, it may result in exponentially-many actions on the number of methods. HGNs are more in line with efforts combining HTN and standard planning approaches (Kambhampati, Mali, and Srivastava 1998; Estlin, Chien, and Wang 1997); the main difference is that HGNs eliminate the use of tasks all-together while still preserving the expressiveness of Simple Hierarchical Ordered Planning (Shivashankar 2015).

The problem of learning hierarchical planning knowledge has been a frequent research subject over the years. For example, ICARUS (Choi and 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 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 an 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 an standard planner is used to generate a plan to get the package from $L1$ to $L2$ and skills are used to learn new HTN methods from the plan generated to fill the gap on how to get from $L1$ to $L2$.

Another example is HTN-Maker (Hogg, Muñoz-Avila, and Kuter 2008). HTN-Maker uses task semantics defined as *(preconditions,effects)* pairs, exactly like TMKs mentioned before, 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 such a situation recursion is learned. HTN-Maker learns incrementally after each training case is given.

HTNLearn (Zhuo, Muñoz-Avila, and Yang 2014) 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).

Similar to HTN planning, hierarchical decompositions have been used in *hierarchical reinforcement learning* (Parr and 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.

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 fo.rm 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).

Inductive learning has been used to learn rules indicating goal-subgoal relations in X-learn (Reddy and Tadepalli 1997). This is akin to learning macro-operators (Mooney 1988; Botea, Müller, and Schaeffer 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 and Laird 2006). It uses as input annotated behavior trace structures, indicating the decisions that led to the plans; this is used to generate a 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 and Laird 2006), to be annotated with the subgoals as they are accomplished in the traces. In our proposed work, the input traces are not annotated and, more importantly, we are learning HGNs.

Goal regression techniques have been used to generate a plan starting from the goals that must be achieved (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 contrast, in a HGN, the hierarchies of goals represent relations between the HGN methods and are not necessarily implied directly from the actions. Making an analogy with HTN methods, HGN methods capture additional domain-specific knowledge (Nau et al. 2003) or generate plans with desirable properties (e.g., taking into account quality considerations) again not explicitly represented in the actions (Hogg, Kuter, and Munoz-Avila 2010).

Work on learning hierarchical plan knowledge is related to learning of context-free grammars (CFGs), which aims at eliciting a finite set of production rules from a finite set of strings (Oates, Desai, and Bhat 2002; Sakakibara 1997). The precise definition of the learning problem varies constraining the resulting CFG by, among others, (1) providing a target function (e.g., obtaining a CFG with the minimum number of production rules) or (2) assuming that negative examples (i.e., strings that must not be generated by the CFG) are given. To learn CFGs, algorithms search for production rules that generate the training set (and none of the negative examples when provided). Grammar learning is exploited by the Greedy Structure Hypothesizer (GSH) (Li, Kambhampati, and Yoon 2009), which uses probabilistic context-free grammars learning techniques to learn a hierarchical structure of the input plan traces. GSH doesnt learn preconditions since its goals are not to generate the grammars for planning but to reflect users preferences. The difference between learning CFG and learning hierarchical planning knowledge is twofold. First, characters that form a string have no meaning. In contrast, actions in a given plan are defined by their preconditions and effects. This means that plausible strings generated by the grammars may be invalid when viewed as plans. Second, learning HGNs requires not only learning the task decomposition but also the preconditions. This is an important difference: HTNs are strictly more expressive than CFGs (Erol, Hendler, and Nau 1996). Intuitively, HTNs are akin to context-sensitive grammars in that they constraint when a decomposition can take place. Context-sensitive grammars are also strictly more expressive then CFGs (Sipser 2006).

Finally, as we will see in the next the next section, our proposed work is related to the notion of planning landmarks (Hoffmann, Porteous, and Sebastia 2004). Given a **planning problem** P, defined as a triple $(s_0, g, \mathcal{A})$, indicating the initial state, the goals and the actions respectively, 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 identify automatically landmarks for P. Planning landmarks have been widely used for automated planning resulting in planners such as LAMA (Richter and Westphal 2010) and the HGN planner GoDel (Shivashankar et al. 2013).

ND learning problem

We want to learn HGNs for fully observable nondeterministic (FOND) planning (Fu et al. 2011; Speck, Ortlieb, and Mattmüller 2015; Winterer, Mattmüller, and Wehrle 2015). In such domains, actions may have multiple outcomes. For example, in the Minecraft simulation, when a character swings a sword to hit a monster, there are two possible outcomes: either the sword hits the monster or the monster parries it and the sword doesn't hit anything.

As discussed before, HTN learners require the tasks semantics to be given either as Horn clauses defining the tasks or as *(preconditions,effects)* pairs. The latter is used, for example, in the nondeterministic HTN learner ND-HTNMaker, a state-of-the-art HTN learner, to pinpoint locations in the traces where the various tasks are fulfilled. ND-HTNMaker enforces a right recursive structure: exactly one primitive task followed by none or exactly one compound task. The main objective of enforcing this right recursive structure is to deal with nondeterminism: if, for example, the character swings the sword (e.g., a primitive task), the follow-up compound task handles the nondeterminism: one method decomposing a compound task t will simply perform the action to swing

at the monster followed by t, thereby ensuring that method can be triggered as many times as needed until the monster is hit (and dies). Other methods decomposing t handle the case when the monster has been dealt with (e.g., a method handling the case when "character next to a dead monster"). This ensures that methods learned by HTN-Maker are provable correct (Hogg, Kuter, and Muñoz-Avila 2009). Correctness can be loosely defined as follows: any solution generated by a sound nondeterministic HTN planner such as ND-SHOP (Kuter and Nau 2004) using the learned methods and the nondeterministic actions is also a solution when using the nondeterministic actions (i.e., without the methods).

Like in the deterministic case, the inputs will be (1) a collection of actions $\mathcal{A}$ and (2) a collection of traces $s_0\ a_0\ s_1\ a_1\ \ldots\ a_n\ s_{n+1}$, where each $a_i \in \mathcal{A}$. Only this time, any action $a_i \in \mathcal{A}$ may have multiple outcomes; so each occurrence of a_i in the input traces will reflect one such outcome.

Planning in nondeterministic domains requires to account for all possible outcomes. As such, (Cimatti et al. 2003) proposed a categorization of solutions for nondeterministic domains. It distinguishes between weak, strong cyclic and strong solutions for a problem $(s_0, g, \mathcal{A})$. A solution is represented as a policy $\pi : S \rightarrow A$, a mapping from the possible states in the world S to actions A, indicating for any given state $s \in S$, what action $\pi(s)$ to take. Given a policy π, an **execution trace** is any sequence $s_0\ \pi(s_0)\ s_1\ \pi(s_1)\ \ldots\ \pi(s_n)\ s_{n+1}$, where s_i is an state that can be reached from state s_{i-1} after applying action $\pi(s_{i-1})$.

A solution policy π is *weak* if there exists an execution trace from s_0 to a state satisfying g. Weak solutions guarantee that a goal state can be successfully reached sometimes. For example, in the Minecraft simulation, a policy that assumes a computer-controlled character will always hit any monster it encounters when swinging the sword is considered a weak solution. In particular, this solution would not account for the situation when the monster parries the character's sword attack; e.g., the monster might counter-attack and disable the agent and the agent has not planned what to do in such a situation. Under the fairness assumption, stating that *"every action executed infinitely often will exhibit all its effects infinitely often"* (D'Ippolito, Rodrıguez, and Sardina 2018), a solution π is either strong cyclic or strong if (1) every terminal state entails the goals and (2) for every state s that the agent might finds itself in after executing π from s_0, there exists an execution trace from the state s to a state satisfying g. The difference is that in strong cyclic solutions the same state might be visited more than once whereas in strong solutions this never happens. For example, a strong cyclic solution might have the character swing the sword against the monster and if the monster parries the attack, the character takes an step back to avoid the monster's counter-attack and step towards the monster while taking another swing at it; this can be repeated as many times as needed until the monster dies. Strong solutions are ideal since they never visit the same state but in some domains they might not exists. For instance, there are no strong solutions in the Minecraft simulation mentioned as the monster can repeatedly parry the character's attacks. The same occurs in the robot navigation domain (Cimatti et al. 2003), created to model nondeterminism. In this domain a robot is navigating between offices and when it encounters a closed door for an office it wants to access, the robot will open it. There is an another agent acting in the environment that closes doors at random. So the robot might need to repeatedly execute the action to open the same door.

Solving nondeterministic planning problems is difficult because of what has been dubbed the explosion of states as a result of the nondeterminism (Fu et al. 2011). One demonstrated way to counter this is by adding domain-specific knowledge as described in (Kuter and Nau 2004). While the algorithm described is generic for a variety of ways to encode the domain-specific knowledge, it showcases hierarchical planning techniques outperforming an state-of-the-art nondeterministic planner in some domains including the robot navigation domain. The results show either speedups of several orders of magnitude or the ability to solve problems of sizes, measured by the number of goals to achieve, previously impossible to solve.

Relation to probabilistic domains. In this work we are neither assuming a probability distribution over the possible actions' outcomes to be given nor we aim to learn such a distribution. Once an HGN domain is learned, hierarchical reinforcement learning techniques (Dietterich 2000) can be used to learn a probability distribution over the various possible goal decompositions and exploit the learned distribution during problem solving as done in (Hogg, Kuter, and Munoz-Avila 2010).

We propose to learn bridge atoms and their hierarchical structure with the important constraint that the learned hierarchical structure must encode the domain's nondeterminism in a sound way. For instance, the nondeterministic version of the logistics transportation domain in (Hogg, Kuter, and Muñoz-Avila 2009) extends the deterministic version as follows: when loading a package p into vehicle v in a location l there are two possible outcomes: either p is inside v or p is still at l (i.e., the load action failed). Regardless of possibly repeating the same action multiple times, traces will bring the package to the airport, transport it by air to the destination city, and deliver it. So the kinds of decompositions we are aiming to learn should also work on nondeterministic domains; on the other hand a learned hierarchy would be unsound if, for example, it assumes that the load truck action always succeeds and immediately proceeds to deliver the package to an airport. This will lead to weak solutions.

To correctly handle nondeterminism, we propose forcing a right-recursive structure on lower echelons of the learned HGNs. This takes care of the nondeterminism and combine well with the higher decompositions. For instance, in the transportation domain we identified a goal g_{airp}, for the package p reaching the airport, identified as a bridge atom, and then have all methods achieving g_{airp} be right recursive; e.g., methods of the form $(:\ method\ g_{airp}\ prec\ (g\ g_{airp})\ <)$, where g is some intermediate goal such as loading the package into a vehicle.

Defining the Learning Problem

Our aim is the automated learning of HGN methods. This includes learning the goals, the goal-subgoal structure of the HGN methods and their applicability conditions. Specifically, the learning problem can be defined as follows: given a set of actions $\mathcal{A}$ and a collection of traces Π generated using actions in $\mathcal{A}$, to obtain a collection of HGN methods. A collection of methods $\mathcal{M}$ is correct if given any *(initial state, goal)* pair (s_0, g), and any solution plan π generated by a sound HGN planner using $\mathcal{M}$ and $\mathcal{A}$, π is a correct plan solving the planning problem $(s_0, g, \mathcal{A})$. An HGN method m is a construct of the form *(:method head(m) preconditions(m) subgoals(m) <(m))* corresponding to the goal decomposed by m (called the head of m), the preconditions for applying m and the subgoals decomposing *head(m)*. Figure 1 shows an example of an HGN method in the logistics transportation domain (Veloso 1994). (the question marks indicate variables. It recursively decomposes the goal of delivering *?pack1* into *?loc2* into three subgoals: (1) delivering *?pack1* to the airport *?airp1* in the same city as its current location *?loc1*, (2) delivering *?pack1* to the airport *?airp2* in the same city as the destination location *?loc2*, and (3) recursively achieve the head goal):

Head: Package-delivery *Preconditions:* (at ?pack ?loc1 ?city1) (airport ?airp1 ?city1) (airport ?airp2 ?city2) (location ?loc2 ?city2) ($\neq$?city1 ?city2) *Subgoals:* g_1: (package-at ?pack ?airp1) g_2:(package-at ?pack1 ?airp2) g_3:(package-at ?pack ?loc2 ?city2) *Constraints:* $g_1 < g_3$, $g_2 < g_3$

Figure 1: Example of an HGN method in the logistics transportation domain. The question marks indicate variables. The goal achieved by the method is the last subgoal, g_3. It recursively decomposes the goal of delivering *?pack* into *?loc2* into three subgoals: (1) delivering *?pack1* to the airport *?airp1* in the same city as its current location *?loc1*, (2) delivering *?pack* to the airport *?airp2* in the same city as the destination location *?loc2*, and (3) g_3 is to be achieved after g_1 and g_2 are achieved.

HGNs planners (Shivashankar et al. 2013; 2012) maintain a list $G = \langle g_1, \ldots, g_n \rangle$ of open goals (i.e., goals to achieve). Planning follows a recursive procedure, starting with $\pi = \langle\rangle$, choosing the first element, g_1, in G, and either (1) applying an HGN method m decomposing g_1 into m's subgoals $\langle g'_1, \ldots, g'_k \rangle$, concatenating m's subgoals into G (i.e, $G = \langle g'_1, \ldots, g'_k, g_1, \ldots, g_n \rangle$ are the new open goals), or (2) applying an action $a \in \mathcal{A}$ achieving g, appending a to π (i.e., $\pi \leftarrow \pi \cdot a$) and removing g from G. In either case it will check if the preconditions of m (respectively, a) are satisfied in the current state. When a is applied, the current state is transformed in the usual way (Fikes and Nilsson 1971). When $G = \emptyset$, π is returned. HGN planners extend this basic procedure to allow the use of standard planning techniques to achieve open goals and to enable a partial ordering between the methods' subgoals. the planner picks the first goal in G without predecessors. For example, in Figure 1, the user may define the constraints: $g_1 < g_3, g_2 < g_3$, and the planner instead of always picking the first subgoal in G, it picks the first subgoal without predecessors. [1]

Learning Hierarchical Goal Structures

We propose transforming the problem of identifying the goals and learning their hierarchical relation into the problem of finding relations between word embeddings extracted from text. Specifically, we propose viewing the collection of input traces Π as text: each plan trace $\pi = s_0\ a_0\ s_1\ a_1\ \ldots\ a_n\ s_{n+1}$ is viewed as a sentence $w_1\ w_2\ \ldots\ w_m$; each action a_i and each atom in s_j is viewed as a word w_k in the sentence. The order of the plan elements in each trace is preserved (we use the term **plan element** to refer to both atoms and actions): the word $w_j = a_i$ appears before the word $w_{j'} = p$, for every $p \in s_{i+1}$. In turn, every $w_{j'}$ appears before $w_{j''} = a_{i+1}$.

Word embeddings are vectors representing words in a multi-dimensional vector space (Bengio et al. 2003; Baroni, Dinu, and Kruszewski 2014). There are a number of algorithms to do this translation (Mikolov et al. 2013; Pennington, Socher, and Manning 2014). They have in common that they represent vector similarity based on the co-occurrence of words in the text. That is, words that tend to occur near one another will have similar vector representations. In our preliminary work we used Word2Vec (Mikolov et al. 2013) (i.e., Word-Neighboring Word), a widely used algorithm for generating word embeddings. Word2Vec uses a shallow neural network, consisting of a single hidden layer, to compute these vector representation; it computes a context window $\mathcal{W}$ consisting of k contiguous words and trains the network using each word $w \in \mathcal{W}$ (i.e., $\mathcal{W}$ is w's context). The window $\mathcal{W}$ is "moved" one word at the time through the text further training the network each time. Training is repeated with windows of size $i = \{1, 2, \ldots k\}$. For this reason, Word2Vec is said to use "dynamic windows". In Word2Vec, similarity is computed with the cosine similarity, sim_C, because it measures how close is the orientation of the resulting vectors, which are distributed in such a way that words frequently co-occurring in the context windows have similar orientation whereas those that co-occur less frequently will have a dissimilar orientation.

There are two particularities of the change of representation from plan elements to word embeddings that is particularly suitable for our purposes: first the procedure is unsupervised. This means in our case that we do not have to annotate the traces with additional information such as where the goals are been achieved in the traces. Second, vector representations are generated based on the context in which they occur (e.g., the dynamic window $\mathcal{W}$ in Word2Vec). In our case, the vector representations of the plan elements will be generated based on their proximity to other plan elements in the traces. These vectors can be clustered together into plan elements that are close to one another.

[1] Actions are *(preconditions,effects)* pairs, where the effects consist of the add- and the delete-lists of atoms. If the preconditions are satisfied in the current state, the state is transformed as indicated by the effects: atoms in the delete-list are removed and atoms in the add-list are added.

Our working hypothesis, supported by previous work (Gopalakrishnan, Muñoz-Avila, and Kuter 2018), is that what we call **bridge atoms**, are ideal candidates for goals. Given two clusters of plan element embeddings, A and B, a *bridge atom*, $bridge_{AB}$, is an atom in either A or B that is most similar to the plan elements in the other set.

Establishing a bridge atom hierarchy is a recursive process that first requires calculating the bridge atom of a corpus, splitting each text around the bridge atom so that each text in the corpus becomes two new texts, before and after the bridge atom, and then repeating the procedure on the resulting sub-corpora.

The procedure for find a bridge atom for a corpora is as follows. We train a Word2Vec model on the corpus to determine the word vectors and cluster them with Hierarchical Agglomerative Clustering. Currently we limit the number of clusters to two, although later research may explore how to determine the number of clusters from the structure of the traces. We determine the cosine distance of each atom in a cluster to each atom in the other and average them together for each atom, selecting the word with the shortest average distance,[2]

$$bridge_{AB} = argmin_{a \in A, b \in B}(\frac{1}{|B|}\Sigma_{b \in B} dist_C(a,b), \frac{1}{|A|}\Sigma_{b \in B} dist_C(a,b)), \quad (1)$$

where $dist_C$ is the cosine distance between the vector representations of two atoms. If an action is selected as the bridge atom, we instead use in its place the atom describing one of its goals.

As previously stated, by splitting each trace around the bridge atom, we can form two new sub-corpora, one from the section of each trace before the bridge atom and one from the section after the bridge atom. Then we recursively perform the procedure for bridge atom selection on each new corpora, keeping track of the hierarchical relationship of each sub-corpora to the other corpora. If during the division process, a section of a trace becomes shorter than some threshold, we discard it from the sub-corpora. Progress along any branch of recursion halts once there are insufficient traces in a sub-corpus for training.

We use the hierarchy of bridge atoms as a guide for building a set of hierarchical methods. At the lowest level of division are single-action or short multi-action sections of the traces. Each of these sections will become a method with a single goal (an effect of an action) or a method with multiple goals (one for each of the actions). Each of these methods have two subgoals: one for the subsection of trace before a bridge atom and another one for the trace after that bridge atom.

Each action is annotated with its preconditions. The preconditions of a method can be extrapolated from the preconditions of the actions into which it decomposes by regressing over the actions of that section of the plan trace in reverse, collecting the action preconditions and removing from the preconditions any atom which is in the effects of chronologically-preceding action.

[2]This is different from the formula we used in (Gopalakrishnan, Muñoz-Avila, and Kuter 2018), which computed the maximum similarity.

Current Status

We are using a variant of the Pyhop HTN planner (`https://bitbucket.org/dananau/pyhop`). Our variant introduces nondeterminism in the actions and generates solution policies as described in (Kuter and Nau 2004).

Our experiments use a nondeterministic variant of the logistics domain (Veloso 1992). In the domain, packages must be relocated from one location to another. Trucks transport packages within cities, and airplanes transport packages between cities via airports. Nondeterminism is introduced via the load and unload operators, which have two outcomes, success (the package is loaded onto/unload from the specified vehicle) or failure (the package does not move from its original location). We have also added rockets that transport packages between cities on different planets via launchpads. All traces demonstrate a plan for achieving the same goal, the relocation of a package from a location in one city on the starting planet to a location in a city on the destination planet.

To ensure that Word2Vec can identify common bridge atoms across the corpus, the package and each location must have the same name in all traces. Although Word2Vec typically works best on a corpus of thousands of texts or more, we are able to learn reasonable bridge atoms from hundreds of texts by increasing the number of epochs and lowering learning rate. For our problem design, a reasonable first bridge atom is one that involves the package and a rocket or launchpad, as transporting the package from the start planet to the destination planet marks the halfway point in the traces. From a corpus of 700 traces, with 1000 epochs and a learning rate of 0.00025, our first bridge atom is the action unload(package, rocket).

Because word embeddings are sensitive to word context, the trace structure influences the bridge atom hierarchy. Which atoms are included in the trace and where they are included is important. We are experimenting with two different variants of state expression within traces. In one variant, we list each action preceded by its deletelist and followed by its addlist. If an atom occurs in the addlist of one action and the deletelist of the subsequent action, that atom will only appear in the addlist of the first action. In another variant, we list actions preceded by their preconditions and followed by their effects. In both variants, atoms are listed alphabetically.

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

References

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

Alford, R.; Kuter, U.; and Nau, D. S. 2009. Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Baroni, M.; Dinu, G.; and Kruszewski, G. 2014. Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. In *ACL (1)*, 238–247.

Bengio, Y.; Ducharme, R.; Vincent, P.; and Jauvin, C. 2003. A neural probabilistic language model. *Journal of machine learning research* 3(Feb):1137–1155.

Bergmann, R., and 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.; and Schaeffer, J. 2005. Learning partial-order macros from solutions. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 231–240.

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

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

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence* 147(1-2):35–84.

Currie, K., and Tate, A. 1991. O-Plan: The open planning architecture. *Artificial Intelligence* 52(1):49–86.

Dayan, P., and Hinton, G. E. 1993. Feudal reinforcement learning. In *Advances in neural information processing systems*, 271–278.

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

D'Ippolito, N.; Rodrıguez, N.; and Sardina, S. 2018. Fully observable non-deterministic planning as assumption-based reactive synthesis. *Journal of Artificial Intelligence Research* 61:593–621.

Diuk, C.; Schapiro, A.; Córdova, N.; Ribas-Fernandes, J.; Niv, Y.; and 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*. Springer. 271–291.

Dvorak, D. L.; Amador, A. V.; and Starbird, T. W. 2008. Comparison of goal-based operations and command sequencing. In *Proceedings of the 10th International Conference on Space Operations*.

Dvorak, D. D.; Ingham, M. D.; Morris, J. R.; and Gersh, J. 2009. Goal-based operations: An overview. *JACIC* 6(3):123–141.

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

Erol, K.; Hendler, J.; and Nau, D. S. 1996. Complexity results for hierarchical task-network planning. *Annals of Mathematics and Artificial Intelligence (AMAI)* 18:69–93.

Estlin, T. A.; Chien, S.; and Wang, X. 1997. An argument for a hybrid HTN/operator-based approach to planning. In *European Conference on Planning (ECP)*, 184–196.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Fu, J.; Ng, V.; Bastani, F. B.; Yen, I.-L.; et al. 2011. Simple and fast strong cyclic planning for fully-observable nondeterministic planning problems. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, 1949.

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

Gopalakrishnan, S.; Muñoz-Avila, H.; and Kuter, U. 2018. Learning task hierarchies using statistical semantics and goal reasoning. *AI Communications* 31(2):167–180.

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

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

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)* 14:253–302.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *Journal of Artificial Intelligence Research* 22:215–278.

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

Hogg, C.; Kuter, U.; and Munoz-Avila, H. 2010. Learning methods to generate good plans: Integrating htn learning and reinforcement learning. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*.

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

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

Kambhampati, S.; Mali, A.; and Srivastava, B. 1998. Hybrid planning for partially hierarchical domains. In *National Conference on Artificial Intelligence (AAAI)*, 882–888.

Knoblock, C. A. 1994. Automatically generating abstractions for planning. *Artificial intelligence* 68(2):243–302.

Könik, T., and Laird, J. E. 2006. Learning goal hierarchies from structured observations and expert annotations. *Machine Learning* 64(1-3):263–287.

Kuter, U., and Nau, D. S. 2004. Forward-chaining planning in nondeterministic domains. In *National Conference on Artificial Intelligence (AAAI)*, 513–518.

Kuter, U.; Sirin, E.; Nau, D. S.; Parsia, B.; and Hendler, J. 2005. Information gathering during planning for web service composition. *Journal of Web Semantics (JWS)* 3(2-3):183–205.

Li, N.; Kambhampati, S.; and Yoon, S. W. 2009. Learning probabilistic hierarchical task networks to capture user preferences. In *IJCAI*, 1754–1759.

McDermott, D. V. 2002. Estimated-regression planning for interactions with web services. In *AIPS*, volume 2, 204–211.

Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G. S.; and Dean, J. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, 3111–3119.

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

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

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

Murdock, J. W., and Goel, A. K. 2001. Meta-case-based reasoning: Using functional models to adapt case-based agents. In Aha, D. W.; Watson, I.; and Yang, Q., eds., *Fourth International Conference on Case-Based Reasoning*.

Murdock, J. W. 2001. *Self-improvement through self-understanding: Model-based reflection for agent adaptation*. Ph.D. Dissertation, Georgia Institute of Technology.

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

Nau, D. S.; Muñoz-Avila, H.; Cao, Y.; Lotem, A.; and Mitchell, S. 2001. Total-order planning with partially ordered subtasks. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research (JAIR)* 20:379–404.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Muñoz-Avila, H.; Murdock, J. W.; Wu, D.; and Yaman, F. 2005. Applications of SHOP and SHOP2. *IEEE Intelligent Systems* 20(2):34–41.

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

Oates, T.; Desai, D.; and Bhat, V. 2002. Learning k-reversible context-free grammars from positive structural examples. In *International Conference on Machine Learning (ICML)*, 459–465.

Ontanón, S.; Mishra, K.; Sugandh, N.; and Ram, A. 2010. On-line case-based planning. *Computational Intelligence* 26(1):84–119.

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

Pennington, J.; Socher, R.; and Manning, C. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 1532–1543.

Pollock, J. L. 1998. The logical foundations of goal-regression planning in autonomous agents. *Artificial Intelligence* 106(2):267–334.

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

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

Sakakibara, Y. 1997. Recent advances of grammatical inference. *Theoretical Computer Science* 185(1):15–45.

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

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

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

Shivashankar, V. 2015. *Hierarchical Goal Network Planning: Formalisms and Algorithms for Planning and Acting*. Ph.D. Dissertation, Dept. of Computer Science, University of Maryland.

Sipser, M. 2006. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston.

Speck, D.; Ortlieb, M.; and Mattmüller, R. 2015. Necessary observations in nondeterministic planning. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, 181–193. Springer.

Tao, F.; Zhao, D.; Hu, Y.; and 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(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. In *LWA*, 74–79.

Veloso, M. M. 1992. Learning by analogical reasoning in general problem solving. PhD thesis CMU-CS-92-174, School of Computer Science, Carnegie Mellon University.

Veloso, M. M. 1994. *Planning and learning by analogical reasoning*. Springer-Verlag.

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

Wilkins, D., and desJardins, M. 2001. A call for knowledge-based planning. *AI Magazine* 22(1):99–115.

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

Winterer, D.; Mattmüller, R.; and Wehrle, M. 2015. Stubborn sets for fully observable nondeterministic planning. In *ICAPS*. AAAI Press.

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