Learning Task Hierarchies Using Statistical Semantics and Goal Reasoning

Sriram Gopalakrishnan
Lehigh University, CSE, 19 Memorial Drive West, Bethlehem, PA 18015-3084 USA
E-mail: srg315@lehigh.edu

Héctor Muñoz-Avila
Lehigh University, CSE, 19 Memorial Drive West, Bethlehem, PA 18015-3084 USA
E-mail: munoz@cse.lehigh.edu

Ugur Kuter
SIFT, LLC, 9104 Hunting Horn Lane, Potomac, Maryland 20854 USA
E-mail: ukuter@sift.net

This paper describes WORD2HTN, a new approach for learning hierarchical tasks and goals from plan traces in planning domains. Our approach combines semantic text analysis techniques and subgoal learning in order to produce Hierarchical Task Networks (HTNs). Unlike existing HTN learning algorithms, our system uses semantics and similarities of the atoms and actions in the plan traces. WORD2HTN first learns vector representations that represent the semantics and similarities of the atoms occurring in the plan traces. Then the system uses those representations to group atoms occurring in the traces into clusters. Clusters are grouped together into larger clusters based on their similarity. These groupings define a hierarchy of atom clusters. These atom clusters help to define task and goal hierarchies, which can then be transformed into HTN methods and used by an HTN planner for automated planning. We describe our algorithm and present our experimental evaluation.

Keywords: Goal Reasoning, Learning Goal Structures, Learning HTNs, Word Embeddings, Statistical Semantics, Clustering

Introduction

Hierarchical Task Network (HTN) planning is a problem-solving paradigm for generating plans (i.e., sequences of actions) that achieve specified tasks. Tasks are symbolic representations of activities such as `achieve goal g`, where *g* is an standard goal. Tasks can be more abstract activities such as `patrol an area`. An HTN planner recursively generates a plan by decomposing tasks of higher level of abstraction into simpler, more concrete tasks. The task decomposition process terminates when a sequence of actions is generated. The theoretical underpinnings of HTN planning are well understood [5] and this planning paradigm has been shown to be useful in many practical applications [13,9,2,12].

HTN planners require primarily two knowledge artifacts: planning operators (generalized definitions of actions) and methods (descriptions of how and when to decompose a task into sub-tasks). While operators are generally accepted to be easily elicited in many planning domains, methods require a more involved and time-consuming knowledge acquisition effort: crafting methods requires reasoning about how to combine operators to achieve the subgoals and tasks. Thus automated learning of HTN methods has been the subject of frequent research over the years. Most of the work on learning HTNs use structural assumptions and additional domain knowledge explicitly relating tasks and goals to make goal-directed inferences to produce hierarchical relationships [3,7,23].

This paper describes a new learning method, called WORD2HTN, for learning hierarchical goal relationships (to subgoals) and HTN methods in a planning domain as well as the tasks and their decomposition that accomplish those goals. WORD2HTN takes plan traces as a sequence of words, where each word can be an action or atom. The algorithm takes as input a set of sentences that represent plan traces and uses semantic text analysis to find clusters of atoms, and actions based on semantic similarities. We developed a new hierarchical task learning approach that identifies sub-goals in the input plan traces from the outcomes of semantic-based clustering.

We report our implementation of the WORD2HTN approach and experimentally demonstrate it in a logistics planning domain. In our experiments, we measure

AI Communications
ISSN 0921-7126, IOS Press. All rights reserved

the maximum tree depth of the task hierarchies that generate the solutions for our test problems with the learned HTN methods. We compare the maximum tree depth of the HTN methods by WORD2HTN with trees made from right-recursive combination of actions in the optimal solutions to the test problems. HTN maximum tree depth is an important measure that shows how well the learning process captures goal-subgoal structures and hierarchies. The maximum tree depth is a measure that would be longer (worse) if the learning process failed to divide the goal into appropriate subgoals. If it was imbalanced, then many of the subgoals or tasks would end up on one branch and the maximum tree depth would be longer. The goal structure was determined from *bridge atoms* (which will be formalized in the following preliminaries section).

Our results show that after being trained on a small (not exhaustive) set of plan traces, WORD2HTN learns HTN methods that: (1) Solve all of our randomly-generated test problems and (2) have a shorter maximum tree depth compared to those in right-recursive task hierarchies. The results demonstrate not only efficient planning, but also a more effective learning of goal structure.

1. Preliminaries

We use standard definitions from the planning literature, such as objects, atoms, actions, and state variables [6]. We will refer to these constructs as *planning words*. They correspond to the words in the plan traces. We summarize our definitions for planning words below.

An *action* is a primitive task with arguments that are constant (e.g., with no variables). An action can change the state of the problem when it is applied. Each action has preconditions and effects. We use state-variable representations. The preconditions and effects of an action are sets of atoms. For example, `Move Truck1 Location1 Location2` is an action that has the precondition `Truck1 in Location 1` and the effect `Truck1 in Location2`.

An *atom* is a literal, such as `Truck1 in Location2`. All the literals in this work were atoms. These atoms indicate changes in the variables such as from `Truck1 in Location2` to `Truck1 in Location3`.

A *state* is a collection of literals, indicating the current values for properties of the objects in the domain. A *goal* is a conjunction of literals. A *plan trace* for a state-goal (s,g) pair is a sequence of actions that, when applied in s, produces a state that satisfies the goal g. We represent a plan trace, used for training, as a sequence of *(preconditions, action, effects)* triples: $p_0\ a_0\ e_0\ p_1\ a_1\ e_1 \ldots\ p_n\ a_n\ e_n$, where p_i and a_i are the preconditions and effects of a_i. Usually (e.g., [7,3]), HTN learners represent plan traces as sequences of state-action pairs: $s_0\ a_0\ s_1\ a_1\ \ldots\ a_n\ s_{n+1}$. These HTN learners also assume the actions are given. Without loss of generality, we are simply taken the additional step of identifying for each a_i, how its preconditions are achieved in s_i and how its effects are satisfied in s_{i+1}. We found that by trimming the conditions from the states (i.e., those not mentioned in the preconditions or effects of the immediately preceding or succeeding action), WORD2HTN learns more accurate HTN models.

As an example, the plan trace to transport `package1` from `Location1` to `Location2` could be as follows:

Preconditions `package1 in Location1,`
`Truck1 in Location1`
Action `Load package1 Truck1 Location1`
Effects `package1 in Truck1,`
`Truck1 in Location1`

Preconditions `Truck1 in Location1,`
`Truck1 canReach Location2`
Action `Move Truck1 Location1 Location2`
Effects `Truck1 in Location2`

Preconditions `package1 in Truck1,`
`Truck1 in Location2`
Action `Unload package1 Truck1 Location2`
Effects `package1 in Location2,`
`Truck1 in Location2`

In the previous example, the goal only consists of a single atom, namely `Package1 in Location2`. If the goal also requires `Truck1` to be in `Location2`, then the goal would be the conjunction of two atoms `Package1 in Location2` and `Truck1 in Location2`. In the actual plan traces, that were fed into our algorithm, there was no labeling of components as actions, preconditions, and effects. These were added in the example for readability.

An **Annotated Action Sequence** (AAS) is a triple (π, p, e), where π is a sequence of actions, p is the set of preconditions necessary at the initial state for π to execute successfully, and e is the set of effects in the final state that result from the actions in the plan. p and e

must satisfy the following two conditions: (1) The first action in π is applicable in p, the second action in π must be applicable to the set of atoms resulting from applying the first action in π to p, and so forth. (2) e is the resulting set of atoms after applying the last action in π. Our algorithm automatically infers these annotated sequences either directly from the input traces or by joining previously inferred AAS.

Our algorithm uses a clustering procedure on the atoms occurring in the traces, grouping them and identifying what we call *bridge atoms*. Given two collections of atoms A and B, and a similarity metric *sim* on pairs of atoms, a `bridge atom` a_{AB} between A and B is an atom in either A or B that is most similar to the atoms in the other set as per the following definition:

$$a_{AB} = max(argmax_{a\in A}\Sigma_{b\in B} sim(a,b), \\ argmax_{b\in B}\Sigma_{a\in A} sim(a,b)) \quad (1)$$

The term bridge atom was chosen to reflect that a_{AB} is the nearest element approaching the two sets. If more than one atom satisfies this condition, one is chosen arbitrarily.

A *task* is a *representation* of an activity. Formally, we define a *task* as `t(pre, g)` where `t` is a pair of the form `(name args)` such that `name` is a symbol and `args` is a list of variable or constant symbols. pre is a conjunction of literals denoting the *preconditions* that must be met to start the task and g is a conjunction of literals denoting the *goals* that the task achieves when it is accomplished.

A task can be either *primitive* or *nonprimitive*. A primitive task corresponds to a planning operator and a ground instance of it can be executed directly in the world. A non-primitive task is a task that needs to be decomposed to subtasks in order to achieve it.

In this paper, we restrict ourselves to the Ordered Task Decomposition formalism of HTN planning [15], adopted by HTN planners such as SHOP [15] and SHOP2 [14]. An *HTN method* describes how to decompose non-primitive tasks into simpler ones. Formally, a *method* is a triple `m = ( t, pre,subtasks)`, where `t`, the head of the method, is a non-primitive task; `pre` is a conjunction of logical literals denoting the *preconditions* of the method; `subtasks` is a totally-ordered sequence $\langle t_1, t_2, \ldots, t_k\rangle$ such that each t_k is a task. A method m is *applicable* to a state s, if the method's preconditions are satisfied in s. Each subtask's goals can be the subgoals required in order to achieve the goal literals specified in the top-level task that the method tries to achieve by the decomposition.

An *HTN planning problem* is a tuple $(s_0, T, O, \mathcal{M})$. s_0 is the initial state, T is the initial sequence of tasks to be accomplished, O is a finite set of planning operators, and $\mathcal{M}$ is a finite set of HTN methods. A *solution* to an HTN planning problem is a plan π such that T is an abstraction of π given s_0 [15,14].This would mean that the plan π would satisfy all the goal atoms of the tasks in T. If there is a solution for the HTN planning problem P, then we say that P is *solvable*.

We formalize activity learning, i.e., learning both semantics and structures of activities, as a problem of learning HTN methods. Formally, we define an *HTN learning problem* as a tuple of the form $(\mathcal{P}, O, \mathcal{R})$, where $\mathcal{P}$ is a finite set of plan traces, O is the set of planning operators, and $\mathcal{R}$ is the set of all atoms in the domain. A *solution* for an HTN learning problem is a pair $(\mathcal{T}, \mathcal{M})$ where $\mathcal{T}$ is the set of semantic tasks learned for the planning domain and $\mathcal{M}$ is a set of HTN methods generated to solve those tasks.

2. Learning Vector Representations for Plan Components

The WORD2HTN algorithm learns embedded vector representations of the atoms and actions in the input plan traces. For this purpose, we use a technique from Natural Language Processing (NLP), called Word Embeddings. Word embeddings are vector representations of a word in an N-dimensional space. The words from the input sentences (plan traces) are learned as distributed vectors using WORD2VECTOR [11]. Their vector representations are distributed in such a way that words having shared contexts are more aligned, than those that are do not share context. Here, context means the words before and after the word.

WORD2VECTOR [11] uses a single hidden layer neural network that is trained to generate the words' vector representation. WORD2VECTOR takes as its input a corpus of text. the resulting vector are typically represented in N dimensions. N is a parameter, typically set in the hundreds. The actual value can be chosen using heuristics and/or experimentally. Each unique word in the corpus will have a corresponding vector in the N-dimensional space.

To illustrate how atoms and actions as words embeddings might look, let us consider the atoms `Package1 in Location1` and `Package1 in Truck1`. Their respective vectors in a 3-dimensional space could be $v1$

and $v2$ as shown in Table 1 (this is a simplified example; as indicated before vectors have hundreds of dimensions). The dimensions can be seeing as abstract representations of relationships. So if two vectors have close values in one dimension, then it indicates a degree of similarity or relationship. In this example, all but the last dimension, are close in value. This could happen when the two atoms occur frequently in similar contexts. The WORD2VECTOR algorithm would slowly adjust their vector representations from their initial random values by adjusting their values closer to one another.

When the dimensions of the vector representations of two embeddings (e.g., $v1$ and $v2$) are closely aligned, their cosine similarity is higher. The cosine similarity is defined as the dot product of the normalized vectors as in Equation 2. In the example, $v1$ and $v2$ have close values on two dimensions but not in the third. This can happen, for example, when they co-occur in many but not all of the contexts where either one occur. As a result, their embeddings are pulled in different directions.

The dimensions of the vector representations have no specific meaning or interpretation. Instead, they implicitly encode relationships of the problem space.

$$CosineSim = \frac{\vec{v1} \cdot \vec{v2}}{\|v1\|\|v2\|} \quad (2)$$

Using the cosine as a measure of similarity is standard in Natural Language Processing. A higher value of similarity would result if both words (atoms or actions) are found in many similar contexts. For example, if many traces have the `Package1 in Location1` and then the load-truck action is executed to result in `Package1 in Truck1`, then those two words share a lot of context. So, their vector representations will be closer together.

The most important parameters needed by WORD2VECTOR are: the number of dimensions N, the context window size C, and the learning rate α. We describe how we set these parameters for our experiments in the Evaluation section. For now, it is enough to know what these are to understand how WORD2VECTOR is used by WORD2HTN. We pass

Vector	dim1	dim2	dim3
v1	0.296	0.710	-0.355
v2	0.221	0.774	0.276

Table 1
Example Vector Representations

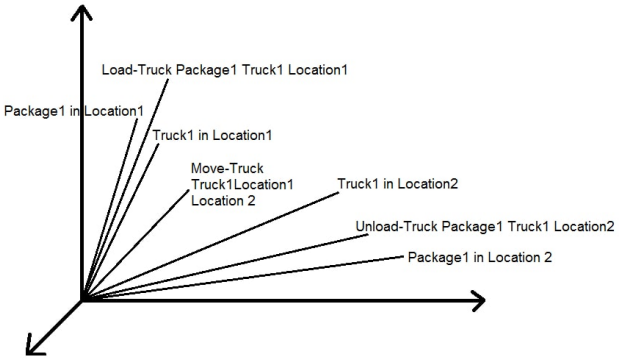

Fig. 1. Sample Distributed Vector Representations of Components from Logistics Domain

into WORD2VECTOR, the plan traces viewing each atom or action as a word. WORD2VECTOR first converts every word of the sentence to be encoded using one-hot encoding. This means every word is converted into a long vector whose size V is the total number of unique words in the data set (the vocabulary of the data). Only one dimension of the long vector for each word in one-hot encoding is set to 1. The vector position set to 1 is unique for every word and represents that word. This is the input form passed into the neural network. The edge weights of the neural network are randomized at the start. During each iteration of the neural network during training, the neural network adjusts the edge weights by the specified learning rate α. The updates are done such that the target word and those within its context have closer vector representations. How this is done and represented in the neural network depends on which architecture is used. There are two common shallow neural network structures in WORD2VECTORḞirst is Continuous-Bag-of-Words (CBOW) and the other is Skip-Gram. [11]. These are two ways of relating a word to it's context, and that determines the network structure We use the Skip-Gram architecture of WORD2VECTOR in our WORD2HTN algorithm (see Figure 2; the left is the CBOW and the right is the Skip-gram structure).

The input for Skip-Gram is the single target word x_k, and the output of the neural network is compared to the each of the words that makes up the current context in which the target word was found $(y_{1k}, y_{2k}, ...y_{Ck})$. So the single input word's representation in the N-dimensional model should be close to the representations of the words in it's context. The error is back propagated to updated the edge weights.

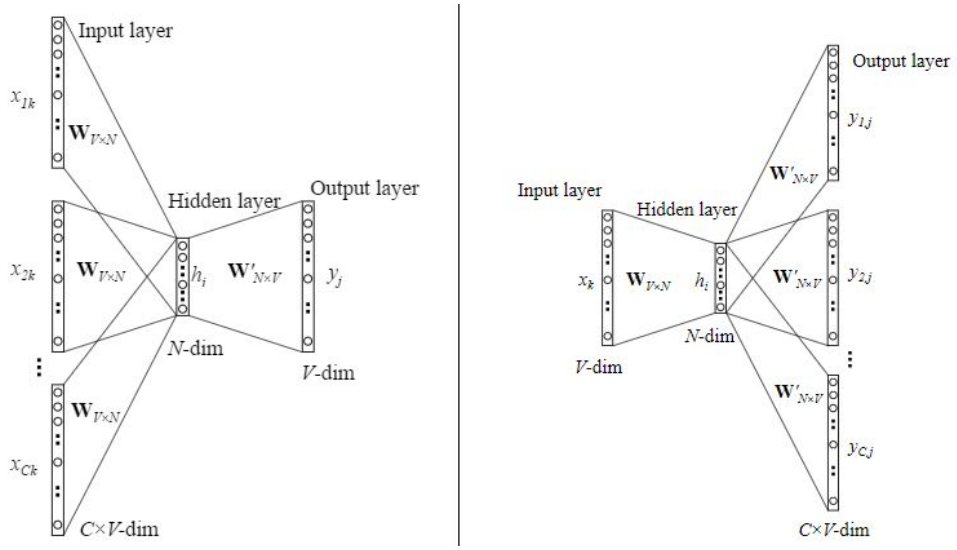

Fig. 2. Comparison of CBOW and SG Neural Network Configurations for WORD2VECTOR. Figure from [19]; used with author's permission

The working of the neural network can be interpreted and better understood in the following way. First, the neural network projects each word's one-hot encoding onto an N-dimensional space. To understand how this is happening, let us look at a simple and smaller network. Let us say that the hidden layer has $N = 3$ nodes, and this means the model is building vector representations of 3 dimensions. For our example, let us set the vocabulary size V as 5. The value of each node in the hidden layer, is the value in one of the N dimensions. The input edge weights are of dimensions 5x3. Each column can be seen as one of the basis vectors of the N-dimensional space, but in the dimensional space of the input (one-hot encoding space). For example, the first dimension's basis vector representation in the V-dimensional space of the input could be of the form $[0.1, 0.02, 0.5, 0.3, 0.7]$. The input word(s) is in one-hot encoding format, so `Package1 in Truck1` could be [0,1,0,0,0]. When an input word is multiplied with the edge weights, what we are really doing is taking the dot-product of the input word, with each of the basis vectors of the N-dimensional semantic space. This is the projection of the input word onto each of the basis vectors. So the values of the nodes $h1, h2, h3$ in our example would be the representation of the word `Package1 in Truck1` in the 3 dimensional space.

In SG network, the hidden layer's values would be the projections of the input word. Then, on the output side, the edge weights represent the N-dimensional representation of each word of the vocabulary. So, column 1 of the output weights would be the vector representation for word 1 of the vocabulary. In our simplified example, the edge weights would be in a 3x5 matrix. So when the hidden layer's values are multiplied by the output edge weights, we are taking the dot product of a vector in the 3-dimensional model with the 3-dimensional representation of each of the words in the vocabulary. If two vectors are similar (closer together), the dot product will be greater. So in our example, the output could be a 5-dimensional vector like [0.2, 0.7, 0.8, 0.1, 0.1], indicating that the model predicts a higher similarity with the vocabulary word at positions 2 and 3. Finally, we run a softmax function on the output to get the likelihood of each output word being semantically related (by being co-contextual). In Skip-Gram, we compare the expected output with the actual result which is the words in the context window. So if the context window is 3 words wide, then we compare the output to the one-hot encoding of the 3 words. The error is the averaged difference between the expected output and the one-hot encoding of each of the words. The error is then back-propagated to updated the edge weights.

At the end of training the WORD2VECTOR component of our algorithm, the output is the set of vector representations of the words. We can analyze them for relationships. As stated, the similarity between two words is defined by cosine similarity (1). To get a visual idea of how vector alignment and similarity would look like, see Figure 1 again, which illustrates how similar vectors will be more closely aligned, and thus have a greater cosine similarity (dot-product) of their vectors.

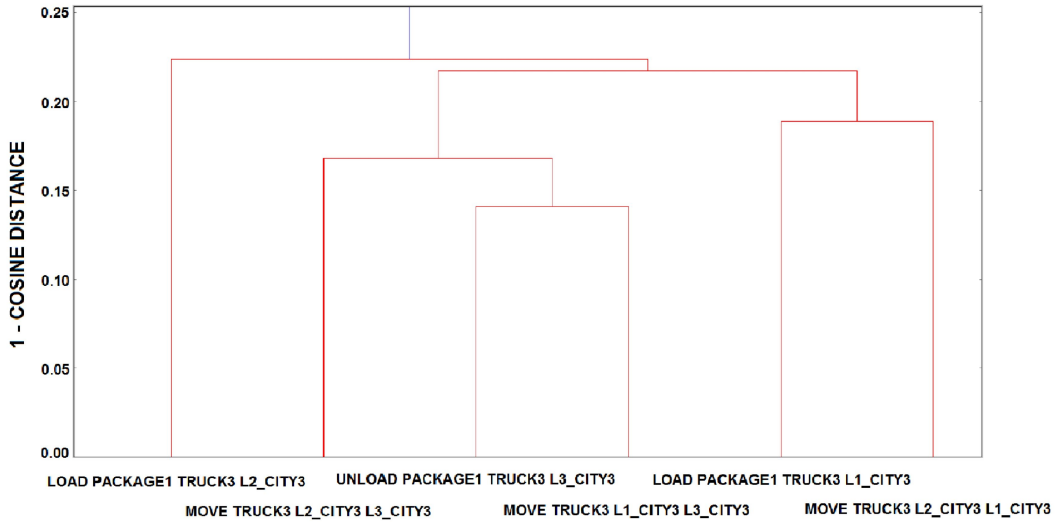

Fig. 3. Hierarchical Agglomerative Clustering of actions within a city for Logistics domain. Some atoms and objects are removed from the x-axis for visibility

3. Learning Goal and Action Hierarchies

After the vector representations for the state atoms and actions in the planning traces are learned, WORD2HTN performs *Hierarchical Agglomerative clustering* [22] of all of the atoms and actions by their semantic similarity (cosine distance). *Hierarchical Agglomerative clustering (HAC)* is a bottom-up clustering method where vectors and groups of vectors (called clusters) are repeatedly grouped with other clusters based on their similarity until a final cluster of all elements is reached. Vectors and groups of vectors that have a higher cosine similarity (lesser cosine distance) to each other, are grouped/clustered together. Less similar vectors and clusters of vectors are grouped later. The result of HAC clustering is a *linkage matrix* that stores the information of how the vectors were clustered together and the hierarchy of clusters thus formed. For example, if the first two atoms to be grouped together are `Package1 in Location1` and `Package1 in Truck1`, then the first row of the linkage matrix will have the ids of these atoms, the cosine distance, and the id of the cluster that the atoms were put into. The second row of the linkage matrix will contain the pair of the second closest atoms, and so on. The algorithm will also merge clusters of atoms into larger clusters. The algorithm groups clusters by using the minimum distance between any word of one cluster with that of the other. The last entry of the linkage matrix contains a cluster with all the atoms and actions.

Since the linkage matrix stores the hierarchy of atom and action clustering, it is used in identifying bridge atoms, which in turn is used to elicit the tasks and subtasks. We will describe this process in Section 4.

Figure 3 illustrates part of the hierarchy learned by HAC in WORD2HTN for the Logistic domain. Only the operators are shown for the sake of clarity. The hierarchy learned from the plan traces, matched what we expected from the training data based on the problem space. For example, all the actions relating to transporting the package within a city (by truck) were all grouped together. The actions related to moving the package between cities (by airplane) were in another lower level cluster.

4. From Action and Atom Hierarchies to HTN Methods for Planning

The algorithm in Figure 4 describe the steps for learning HTN methods from vector representations. It receives as input a collection of traces T. From these traces, WORD2HTN learn the embeddings E of the actions and atoms in the traces. Using the traces and embeddings, the action annotated sequences (AAS) are learned (Line 2). Finally from the learned AAS, WORD2HTN can extract the methods (lines 3 and 4).

The algorithm in Figure 5 learns the AAS. Lines 1 to 6 handle the base case which we will discuss later. The following is an overview of the steps to calculate AAS:

```
LEARNHTNS(T)
1  E ← getEmbeddings(T)
2  AAS ← GETALLANNOTATEDSEQUENCES(T, E)
3  for each k ∈ AAS do
4     MakeMethods(k,∅)
```

Fig. 4. High-level algorithm for learning the HTNs. It receives as parameter the collection of training traces, T

```
GETALLANNOTATEDSEQUENCES(T, E)
 1  if numberOfActions(T) = 1 then
 2     a ← getAction(T)
 3     annSeq ← (a,a.preconditions,a.effects)
 4     return annSeq
 5  elseif (numberOfActions(T) == 0 ) then
 6     return ∅
 7  L ← HAC(T,E)
 8  B ← getBridgeAtoms(L)
 9  for each b ∈ B
10     currentTraces ← findTraces(b)
11     if currentTraces ∈ SeenTracesSet
12        continue; \\back to for loop
13     SeenTracesSet = SeenTracesSet ∪ currentTraces
14     (Ltraces, Rtraces) ←
          splitTraces(currentTraces,b)
15     allLeft ← getAllAnnotatedSequences(Ltraces,E)
16     allRight ← getAllAnnotatedSequences(Rtraces,E)
17     allAnnSeq ← ∅
18     for (La,Ra) ∈ allLeft × allRight
19        if noConflict(La,Ra) then
20           n ← merge(La,Ra)
21           AllAnnSeq ← AllAnnSeq ∪ n
22  return allAnnSeq
```

Fig. 5. Procure to generate all annotated action sequences (AAS). It receives as input the training traces T and the word embeddings E

1. Using HAC, cluster the actions and atoms using cosine similarity; the goal and sub-goal atoms are clustered in this step as well (line 7).
2. Select a bridge atom from the last clustering step and divide the plan traces that contain this atom (lines 8 and 9). This will result in two sets of sub-traces. One set is traces before the bridge atom occurs (i.e., "to the left"), and the other after the bridge atom occurs (i.e., "to the right"; line 14). The bridge atom is always included in the left trace. Line 10 selects all traces having the bridge atom b. Not all traces may contain the same bridge atom, and those traces that do not contain the bridge atom will be covered in a subsequent iteration of the algorithm (the for loop in lines 9-21). The for loop part of the algorithm is explained later in this section.
3. For the left and right sub-traces, The process is repeated recursively (lines 15 and 16) until the algorithm reaches plan traces with a single action or no action (this is the base case in lines 1 to 6). If there is only one action, then the AAS is constructed as indicated line 3. If there are no actions in the traces, then the algorithm returns an empty AAS (line 6)
4. When the algorithm returns from the base case, it then goes back up the recursion hierarchy and repeatedly compose larger AAS from lower-level AAS (lines 18-22). This composition process repeats until the top-level of the hierarchy is reached.

We now describe the algorithm in Figure 5 in more detail.

HAC clustering. HAC clustering (line 7) is done to determine what actions are semantically related to each other. This grouping aims at dividing the plan trace into meaningful sections. The result of HAC clustering is the linkage matrix L, which was previously described in section 3.

Selecting Bridge Atoms and Division of Traces. From the output of the first step of the HAC clustering, the algorithm can elicit a hierarchy of bridge atoms (line 8) to divide the plan traces into recursively smaller parts. This results in dividing the goal of a plan trace into sub-goals which are achieved in the divided parts of the trace. At this step in the algorithm, it takes a top-down approach to decomposing plan traces with bridge atoms (lines 9 to 14). In order to select the bridge atom, the algorithm starts with the highest/last clustering step in the linkage matrix and look at the two smaller clusters that were combined to make it. From the words in the two smaller clusters U and V, the bridge atom is found (which is an atom or an action in the traces) a_{UV} using cosine similarity as in Equation 1.

The bridge atom a_{UV} is the state atom or action most similar to the words in the other cluster. Such a bridge atom would divide plan traces more evenly (closer to the middle) if there is a clear separation or hierarchy in the domain itself. For our running example of transportation within a city, a bridge atom could be `package1 in Truck1`. It appears in traces

close to `package1 in Location1`, as well as close to `package1 in Location2` and is a bridge atom that the trace has to go through. Another example for a larger plan trace is transporting the package across two cities. The package would have to go through an airport for transporting the package to different cities. The bridge atom would be `package in AirportCity1`. This atom would be similar to the atoms in city1 and those of city2.

Using the bridge atom selected, WORD2HTN divides all of the training plan traces into two sets of sub-traces, those appearing before the bridge atom ("to the left"), and those appearing after ("to the right"; line 14). WORD2HTN always chooses the first instance/appearance of the bridge atom in the traces when dividing the traces (line 14). This was done for simplicity and speed. This worked fine for our experiments, but we think its better to select the median instance of the bridge atom (if there is more than one instance in the same trace). Some plan traces will *not* have the bridge atom, and will be ignored for that iteration of the algorithm. However, the ignored traces will be covered in a later iteration (iteration starting at line 9). After a selected bridge atom divides the plan traces into two sets of subtraces, WORD2HTN perform HAC clustering on the atoms and actions in each of the two sets of subtraces (left and right) separately. This is done when WORD2HTN recursively call the function *GetAllAnnotatedSequences* from lines 15 and 16. In the recursive call, line 7 performs HAC clustering only for the atoms and actions in the subtraces. From the new linkage matrix of the HAC output, the algorithm selects a lower-level bridge atom using the same process as before. This bridge atom will divide the smaller traces into two sets again.

Building Annotated Action Sequences. The recursive trace partitioning is done until the base case is reached, in which all of the remaining plan traces have a single action or no action (only atoms). This is captured in lines 1 to 6 of Figure 5. After the base case is reached, the algorithm creates an AAS (with only one action or none at the lowest level). The action is stored with its preconditions and effects into the AAS data structure (line 3). If there is no action, then nothing is stored in the created AAS (lines 5 and 6). In either case, a possibly empty, AAS to the parent (i.e., the one making the recursive call either in Step 15 or Step 16). When both the left and right child cases have returned non-empty AAS, then the AAS are merged or composed together to make a larger AAS as follows: if either side returns more than one AAS, then the algorithm creates AAS by merging pairs, one AAS from the left and right branches of the decomposition (line 18 to 20). The algorithm chooses one from each branch and see if they can be composed together by first checking for conflicts (line 19).

The larger (composed) AAS will contain actions in order, with the left AAS actions first, followed by the actions from the right AAS. It will also contain the net preconditions and net effects of the actions in it. This would be successful and saved only if there are no conflicts between the preconditions and effects of both the left and right child AAS (line 19). If there are conflicts, that particular composition of left and right action sequences will be dropped, and the algorithm will try other combinations. If no combinations are possible, an empty set will be returned to the parent level. While the AAS are merged into larger ones, the algorithm also keeps track of which specific annotated actions sequences were merged (by a unique ID) to get the larger one. This information is tracked separately in a map, and will help in converting the AAS into methods in the final part of the algorithm.

After merging the AAS returned by the child traces, WORD2HTN checks if there are any other bridge atom from the linkage matrix for the *same* set of traces in the parent level. WORD2HTN looks at the bridge atoms from the next row of the linkage matrix (not just the top/last merge step). A new bridge atom is considered if the traces that it is found in contain a different set of actions and atoms than what was seen in the previous iteration of this trace set (checked in line 11). If it is a different set of traces, then the process will repeat with the new bridge atom to learn a new AAS. If there are no more bridge atoms to be considered, WORD2HTN returns the AAS(s) composed at this level to the parent making the recursive call. The composition process of annotated task sequences is repeated up the hierarchy of divided plan traces. It will terminate when the algorithm reaches the level that contains all the plan traces, and when there are no more bridge atoms to consider.

HTN Methods from AAS. Figure 6 shows The final part of the algorithm is to generate methods from the AAS. The goals of the tasks and subtasks at the parent level can be defined by the common objects across the child action sequences. The AAS at any level contains all the actions, preconditions and effects of its children. Finally the AAS are transformed into semantically-annotated methods for HTN planning. The base case is handled in lines 1 to 4. Lines 5 to 10 identify the common objects across the next lower level's AAS. This is done in lines 5 to 10. The variable k is the action

sequence made by ordering the left child AAS actions first, followed by the right child's actions (line 7). After determining k and C, the algorithm can now get the total effects E and net preconditions P of the current AAS (lines 11 and 12). The *task* of the method made from the current AAS is defined as a string with the prefix *AchieveGoal* followed by the atoms in the effects E but filtered such that only those atoms which are predicates of the objects in C will define the task (line 13). The precondition of the method is also inferred by filtering the total set of preconditions by the objects in C (line 14). The subtasks for the method are the tasks of the child AAS, and are returned when the algorithm calls *MakeMethods* on them (lines 18 and 19). Notice that a separate set of common objects $C2$ are obtained in line 17, which is passed down to the child AAS. These become the $targetObjects$ in the recursive call. The only case when $targetObjects$ are not defined (empty set), is when *MakeMethods* is first called in line 4 of Figure 4. Every subsequent recursive call to *MakeMethods* have the $targetObjects$ specified because the tasks of the lower-level should only reference objects relevant at the parent level. This way of defining subtasks guarantees that there is a method to solve it.

After the subtasks are defined, all the parts of a complete method are defined, and it is added to the library of learned methods (line 20). The task associated with the method is returned in line 21. The base case is when an AAS has no actions or a single action. If it has no action, then an empty task is returned (lines 1 and 2). If there is a single action, then the method returns a task corresponding to the effects of the action (lines 3 and 4). An encapsulating method is made for the single action and is added to our library of methods as well. For the methods, all ground objects are lifted and treated as variables. Instances with the same name, become the same variable. It is also possible for duplicate methods to be learned. These are to be removed in post-processing (after learning the methods).

Figure 7 shows an example method made from AAS. In this example the task is transporting a *Package*1 across locations (*Loc*1 to *Loc*3). The common objects across the two children AAS are *Truck*1, *Package*1, and the destination location *Loc*3. Using the common objects, the task and precondition atoms are filtered and used to define the method's task and preconditions (Figure 7 right). The subtasks transport the package to *Loc*3, and then unload it. Recall that the algorithm defines sub-tasks by filtering the effects of the lower level AAS by the common objects.

MAKEMETHODS(AAS, targetObjects)

```
1  if numberActions(AAS) = 0 then
2     return ∅
3  elseif numberActions(AAS) = 1 then
4     return makeEncapsulatingMethod(getAction(AAS))
5  n ← AAS.getLeftActionSequence()
6  m ← AAS.getRightActionSequence()
7  k ← ExtendActionSequence(n,m); \\k ← nm
8  C ← targetObjects
9  if targetObjects = ∅ then
10    C ← getCommonObject(n,m)
11 E ← getEffects(k)
12 P ← getPreconditions(k)
13 task ← "AchieveGoal(" + filterByObjects(E,C) + ")"
14 pre ← filterByObjects(P,C)
15 subtask ← ∅
16 method ← (:method task pre subtask)
17 C2 ← getCommonObject(n,m)
18 method.subTasks ← method.subTasks ∪
   makeMethod(n, C2)
19 method.subTasks ← method.subTasks ∪
   makeMethod(m, C2)
20 AddToMethodLibrary(method)
21 return task
```

Fig. 6. Algorithm for constructing the methods. It receives as parameter the annotated action sequences, *AAS*, and the target objects, $targetObjects$

Using common atoms to define tasks and subtasks works for the class of problems in which there is a central agent or goal object like a package in transportation, or a single agent (actuator) in a domain. If this is not the case, then the algorithm can still work if goal objects are specified in line 4 of the *LearnHTNs* algorithm in Figure 4. In addition, the goal objects must be passed down (along with the common objects) through the hierarchy of AAS to define the task and preconditions for all methods. This is a possible avenue for future research, and we have not yet tested this.

5. Evaluation

5.1. Experimental Domain

We test WORD2HTN in the logistics transportation domain [21]. In this domain, packages must be transported between locations. These locations can be either in the same city, in which cases trucks are used,

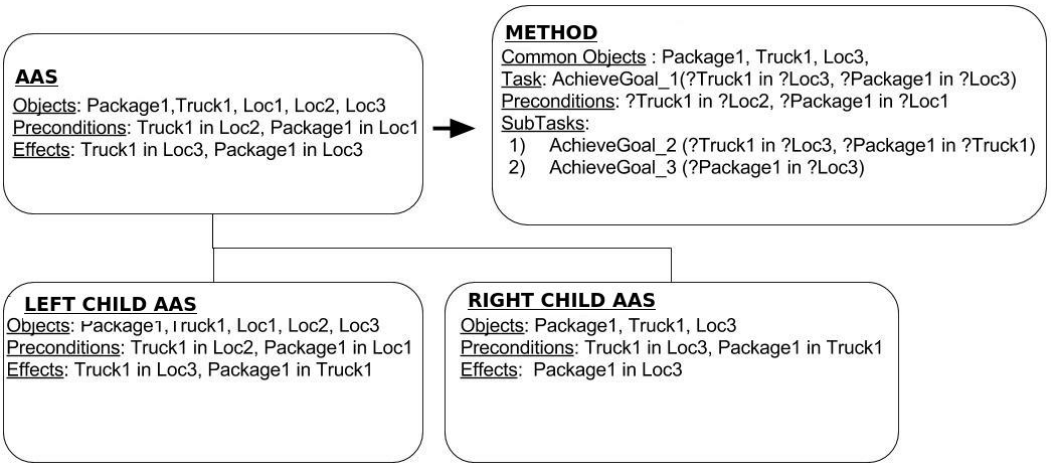

Fig. 7. Converting an AAS to a method based on common objects

or in a different city, in which case airplanes are used. In our experimental setup, we used a problem configuration consisting of 5 cities and 3 locations within each city. We generated plan traces for transporting the package from a randomly-selected location in one of 3 start cities, to a destination location in the remaining 2 cities. We generated the training plan traces for the logistics domain by using a handcrafted set of methods. These plan traces were *optimal solutions*, delivering the packages with the minimum number of actions.

5.2. WORD2VECTOR *Setup*

To train the model of vector representations, we tune parameters of the WORD2VECTOR implementation for it to learn good vector representations. We used the Skip-Gram model of WORD2VECTOR. The most important parameters that were tuned were the following:

Semantic Space Dimensions. The dimension size for the vector representations significantly affects the performance of the model. If there are too few dimensions, the relationships cannot be distributed effectively in the dimensional space. Fewer dimensions would also cause the similarity between vectors to be higher than it ought to be. It is better to have more dimensions, even if the training time increases. As a rule of thumb, we set the dimension size to be twice the number of objects in the domain. The intuition was that every atom and action refers to a subset of the objects in the domain. Therefore, setting the dimension size as a function of the number of objects in the domain would give enough space to distribute the vector representations for effectively capturing the relationships. In our experiments, there were 26 objects, and so we used 52 dimensions. On the other hand, it must be noted that the entire wikipedia corpus (containing a majority of the English vocabulary) was trained with as little as 400 dimensions, and the ensuing model showed strong, expected semantic relationships for the English words. Therefore the number of dimensions is a parameter that need not be set to a high value, and it is likely that 52 dimensions would be enough space to represent relationships for a much larger vocabulary set of atoms and actions.

Context Window Size. The number of words around the target word defines its context. The context window size ought to be large enough to cover the relevant atoms. For our experiments, we chose a window size of 20. The average plan trace length was 102 words (atoms and actions). We chose to err on the smaller side with context size, and iterate over the training data more often. A smaller window size ensures that the atoms and actions that fall inside the context are actually relevant to the target word. A larger window size may gather more noise. If the actions in the plan trace were mixed such that they resulted in the correct final state, but had irrelevant actions interspersed in the plan trace, then it would hurt the learning of relationships. So if the window size is large, it is more likely to pick up noise from the training data that are not semantically relevant. An extreme scenario to illustrate this problem would come from setting the window size is the size of the entire plan trace. In that case, every atom's context is every other atom in the trace and that would be incorrect.

Other parameters. There were other parameters that we adjusted as well. The learning rate was set to 0.001 in our experiments. This was adjusted experimentally. Larger values of the learning rate such as 0.1 and 0.5 did not allow the vector representations to settle into their vector space representations. Large learning rates cause what is called 'overshooting'. Lastly, we iterated over the training data 1000 times to help the vector values converge or settle.

5.3. Experimental Process

The training plan traces were generated from hand coded methods on the logistics domain. We generated 14 different plan traces for training. All of the training traces transport the package from a location in one city, to a location in another city. Each plan trace has a different start and/or goal location. Diverse plan traces help WORD2HTN detect bridge atoms better as they have more information and atoms and actions are more clearly separated from each others' context. These plan traces were *optimal* plan traces for transporting the package. The problem space comprises of 5 cities, with 3 locations in each city. All the plan traces started with the package in one location of a city, with the goal of being transported to another location. Trucks in a city could transport the packages to locations within the city. There was one airport per city. Airplanes can transport packages across cities by flying between airports in those cities.

For testing we randomly generated 30 test planning problems in this domain, by randomly selecting the initial and goal locations of the package for transportation. We ensured that the test problems were unique. That is, no problem used for testing overlap with problems used to generate the training cases.

In our experimental problem setup, there are 225 possible combinations of initial and goal locations for the packages. We verified that the learned methods can cover all cases. This is because of the symmetry and hierarchy inherent in the domain. This also shows that the methods learned were not only capable of solving the problems they were trained from, but general enough to solve problems whose solutions have the same structure or sub-structure as those covered in the training data.

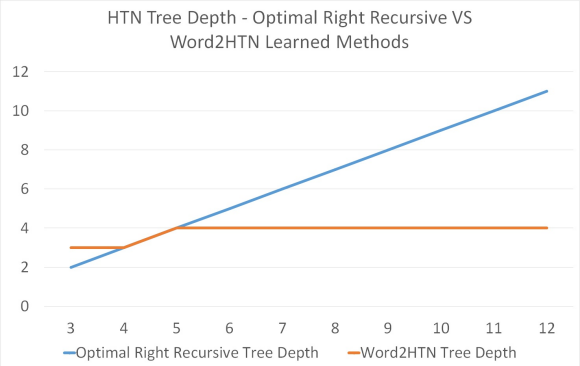

Fig. 8. Comparison of the HTNs learned by WORD2HTN with those in an optimal right-recursive knowledge based on the Maximum Tree Depth measure.

5.4. Results

As a baseline, we used HTN-Maker [7], which is the state-of-the-art HTN learner. HTN-Maker receives as input a collection of traces and the tasks' semantics. We evaluate both approaches with the maximum tree depth of the task hierarchies learned by WORD2HTN and compared them with hierarchical right-recursive task structures that HTN-Maker [7] always generates. Although such right-recursive structures can be suitable for automated planning under different conditions, HTN-Maker cannot capture goal-subgoals structures in the planning domain. On the other hand, WORD2HTN uses bridge atoms to find the semantically related groups of atoms and actions in the problem space and these groups separate subgoals from each other, as well as group together to contain larger goals.

Maximum tree depth is one indicator to measure how well our system captures this knowledge. We compared the maximum tree depth of the HTN methods by WORD2HTN and that of the trees that correspond to an optimal solution from a right-recursive tree decomposition in our test problems.

Figure 8 shows the results comparing the maximum tree depth for different plan lengths. The task decomposition tree depth is shallower than the right recursive tree for longer plans. It is deeper (worse) than the right recursive tree for plan traces of length 3 actions. For 3 actions and less, the tree depth is greater because we have to decompose an additional lower-level task that encapsulates the operator, since every operator is encapsulated in a task. This is an artifact of the way the methods were learned. There are no plan traces shorter than 3 actions because the minimum number of ac-

tions needed to transport a package between one location and the nearest adjacent location is 3 actions. For plan traces of length 5 and larger the tree depth does not increase for WORD2HTN. We are able to achieve a better (shallower) tree depth for larger plan traces because the problem space lends itself to being more evenly decomposed into a hierarchy. In fact our methods captured the structure of the domain, which is what the expert authored methods did as well. What this means in logistics is that transporting a package across cities requires transporting the package to the airport first (clear subtask), which in turn can be decomposed neatly into loading the package into the truck and unloading the package at the airport.

The results thus far do show that the WORD2HTN algorithm was able to capture the structure of the problem space if one exists. There were 37 learned methods compared to the 4 handcrafted methods. However, inspection of the learned methods revealed that these were instances of the more abstract handcrafted methods. The handcrafted methods had more than 3 actions (subtasks) per method, but the learned methods were restricted to 2 subtasks (binary decomposition). Thus more methods were needed to represent the same structure. So indeed the learned methods captured the same structure of the handcrafted methods for the testing domain, albeit represented differently (binary decomposition of tasks). Currently there are a lot of duplicate methods generated. For example the method to transport a package to the airport in city1 maybe identical to the method to transport the package to the airport in city2. Removal of duplicate methods will make the HTN planning faster, and will be addressed in future work. We will also test our algorithm with other domains and trying different settings such as window size, training rates and others.

WORD2HTN was able to solve all of the testing problems, which were different from the problems used to generate the training data.

6. Related Work

The problem of learning HTN planning knowledge has been a frequent research subject. For example, X-LEARN uses inductive generalization to learn task decomposition constructs, which relate goals, subgoals, and conditions for applying these d-rules [18]. ICARUS [3] 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 A and the HTN knowledge is also able to get the package from a location B to its destination, but there is no HTN knowledge on how to get the package from A to B, then a first-principles planner is used to generate a trace to get the package from A to B and skills are used to learn new HTN methods from the generated trace to fill the gap in the HTN learning methods. WORD2HTN learns across multiple traces instead of teleoreactively such as in ICARUS. However, it is conceivable to use teleoreactive techniques assuming there are a few gaps found after the system is deployed. If, however, multiple such gaps are identified, it would be better to learn the domain anew from new and previous traces. This will enable us to take advantage of our system's capability to find common patterns across multiple traces.

Another example is HTN-Maker [7]. HTN-Maker uses task semantics defined as *(preconditions,effects)* pairs to identify sequences of contiguous action sequences in the input plan trace where the preconditions and effects are met. Task hierarchies are learned whenever (1) a task t's preconditions are met before the first action in a trace π and t's effects are met after the last action in π, (2) the preconditions of a task t' are met before the first action in a trace π' and the effects of t' are met after the last action in π', and (3) π is a subtrace of π'. In this situation, t is identified to be a subtask of t'. When $t = t'$ holds, a recursion is learned. HTN-Maker learns incrementally after each training case is given. In contrast, WORD2HTN learns across all traces for common problem-solving patterns. This is part of the reason why in our experiments we are learning more balanced HTNs since our system can learn common or frequently seen problem solving structures.

The only other work that we know which learns HTN planning knowledge across multiple training traces simultaneously to build a model is HTNLearn [23]. HTNLearn transforms the input traces into a constraint satisfaction problem. Like HTN-Maker, it 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.

WORD2HTN does not assume that the task semantics are given. Instead the task semantics in the form of *(precondition, effect)* pairs are learned from the atom clusters elicited from the vector space representations. Another important difference is that HTNLearn is not able to converge to a 100% correct domain (the evaluation of HTNLearn computes the error rates in the learned domain). In contrast, WORD2HTN learns correct domains and in our experiments solve each of the testing problems (which involve relocating a package from an starting location to an ending location).

Similar to hierarchical abstraction techniques used in HTN planning, domain-specific knowledge has been used to search Markov Decision Processes (MDPs) in *hierarchical reinforcement learning* [17,4]. Given an MDP, the hierarchical abstraction of the MDP is analogous to an instance of the decomposition tree that an HTN planner might generate. Given this hierarchical structure, these works perform value-function composition for a task based on the value functions learned over its subtasks recursively. However, the hierarchical abstractions must be supplied in advance by the user. WORD2HTN learns the hierarchical abstractions based on the similarities of the atoms and actions' vector representations.

Hierarchical goal networks (HGNs) [20] 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, general HTN planning is strictly more expressive than HGNs although the total-ordered variant of HTNs has the same expressiveness as HGNs. In [1], a formalism called Goal Task Networks (GTNs) is presented. GTN combines both goal and task networks and it is strictly as expressive as HTNs.

Inductive learning has been used to learn rules indicating goal-subgoal relations in [8]. These rules are used in the SOAR cognitive architecture [10]. Another work on learning goal-subgoal relations is reported in [16]. It uses case-based learning techniques to store goal-subgoal relations, which are then reused by using similarity metrics. Both of these works assume the input action traces to be annotated with the subgoals as they are accomplished in the traces. In our work, the input action traces are not annotated and we are learning HTNs.

7. Conclusions and Future Work

We described WORD2HTN, a new approach for learning hierarchical tasks and HTN methods from plan traces in planning domains. WORD2HTN first learns semantic relationships of atoms and actions from plan traces and encodes them in distributed vector representations using WORD2VECTOR. This vector model is used to cluster the components based on their similarity into hierarchically larger groups using hierarchical agglomerative clustering. This hierarchical grouping is used to identify bridge atoms, and divide the plan traces recursively. The actions are merged into a hierarchy of AAS. The AAS are later converted into methods by using the objects that were common across the lower level AAS or actions. All grounded instances of the same name are treated as the same variable in the learned method.

Our work's premise is particularly suitable for domains that have an underlying hierarchical structure that allows the problem to be decomposed into well-defined subproblems. The logistics transportation domain is an example of such a domain. On the other hand, WORD2HTN may not be suitable for domains that do not have such clear hierarchical structure: examples include domains such as determining if a given input number is a prime number or the puzzle-like domains such as Blocksworld.

Thus far, we have experimented with a deterministic domain. However, we started this research because we think this method can work well with non-deterministic domains since the bridge atoms for problems will still exist. Moving forward, we will generalize our approach to probabilistic actions with multiple outcomes. We intend to generate plan traces for learning from the probabilistic actions, where each plan trace will result from a probabilistic execution of actions. We believe that the rest of the WORD2HTN algorithms will still work with minor modifications because the components with shared contexts will still have similar vector dimensions. One difference is that we believe the similarity of the effects of an action with the action word itself would depend on the frequency of occurrence of the different possible effects. For example if the action `LoadTruck Truck1 Package1` resulted in the atom `Package1 in Truck1` more than 80 percent of the time (over all the training plan traces), then the two components would be more closely related. If 20 percent of the time, it resulted in `Package1 is Broken` then there would be a weaker similarity with the less probable effects. We will investigate this

hypothesis both theoretically and experimentally with non-deterministic plan traces.

We also plan to do a more detailed investigation on the number of dimensions necessary for different domain sizes and types. This analysis would help in selecting the appropriate number of dimensions for a problem, which would make the algorithm faster.

The WORD2HTN algorithm can be parallelized at many points. When we select a bridge atom and split the plan traces about it, then each of the child set of traces can be processed independently of the other before the AAS of the child cases are merged together. Additionally, the process to make methods from AAS can also be parallelized.

Acknowledgments

This research was funded in part by the Office of Naval Research award N00014-18-1-2009, NSF grant 1217888 and by Contract FA8650- 11-C-7191 with the US Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory. Approved for public release, distribution unlimited. The views expressed are those of the authors and do not reflect the official policy of the U.S. Government.

References

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

[2] M. Burstein, R. Goldman, P. Robertson, R. Laddaga, R. Balzer, N. Goldman, C. Geib, U. Kuter, D. McDonald, J. Maraist, P. Keller, and D. Wile. Stratus: Strategic and tactical resiliency against threats to ubiquitous systems. In *Proceedings of SASO-12*, pages 47–54, 2012.

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

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

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

[6] M. Ghallab, D. S. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, May 2004.

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

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

[9] U. Kuter, M. Burstein, J. Benton, D. Bryce, J. Thayer, and S. McCoy. HACKAR: helpful advice for code knowledge and attack resilience. In *Proceedings of AAAI / IAAI*, pages 3987–3992, 2015.

[10] J. Laird. *The Soar Cognitive Architecture*. MIT Press, 2012.

[11] Q. Le and T. Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1188–1196, 2014.

[12] D. Musliner, R. P. Goldman, J. Hamell, and C. Miller. Priority-based playbook tasking for unmanned system teams. In *Proceedings AIAA*. American Institute of Aeronautics and Astronautics, Mar. 2011.

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

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

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

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

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

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

[19] X. Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.

[20] V. Shivashankar, U. Kuter, D. Nau, and R. Alford. 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*, pages 981–988. International Foundation for Autonomous Agents and Multiagent Systems, 2012.

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

[22] J. H. Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

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