WORD2HTN: Learning Task Hierarchies Using Statistical Semantics and Goal Reasoning

Sriram Gopalakrishnan and **Héctor Muñoz-Avila**
Lehigh University
Computer Science and Engineering
19 Memorial Drive West
Bethlehem, PA 18015-3084 USA
{srg315,munoz}@cse.lehigh.edu

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

Abstract

This paper describes WORD2HTN, an algorithm for learning hierarchical tasks and goals from plan traces in planning domains. WORD2HTN combines semantic text analysis techniques and subgoal learning in order to generate Hierarchical Task Networks (HTNs). Unlike existing HTN learning algorithms, WORD2HTN learns distributed vector representations that represent the similarities and semantics of the components of plan traces. WORD2HTN uses those representations to cluster them into task and goal hierarchies, which can then be used for automated planning. We describe our algorithm and present our preliminary evaluation thereby demonstrating the promise of WORD2HTN.

1 Introduction

Hierarchical Task Networks (HTN) planning is a problem-solving paradigm for generating plans (i.e., sequences of actions) that achieve some input tasks. Tasks are symbolic representations of activities such as *achieve goal g*, where *g* is an standard goal. Tasks can be more abstract conditions such as "protect the house". HTN planning generates a plan by using a recursive procedure in which tasks of higher level of abstraction are decomposed into simpler, more concrete tasks. The task decomposition process terminates when a sequence of actions is generated solving the input tasks. HTN planning's theoretical underpinnings are well understood [Erol *et al.*, 1994] and has been shown to be useful in many practical applications [Nau *et al.*, 2005].

HTN planners require two knowledge sources: operators (generalized definitions of actions) and methods (descriptions of how and when to decompose a task). While operators are generally accepted to be easily elicited in many planning domains, methods require a more involved and time-consuming knowledge acquisition effort. Part of the reason for this effort is that crafting methods requires us to reason about how to combine operators to achieve the tasks. For this reason, automated learning of HTN methods have 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 task groupings and hierarchies [Choi and Langley, 2005; Hogg *et al.*, 2008; Zhuo *et al.*, 2014].

In this paper, we present a new semantic goal-reasoning approach, called WORD2HTN, for learning tasks and their decompositions by identifying subgoals and learning HTNs. WORD2HTN builds on work involving word embeddings, which are vector representations for words. These were originally developed for Natural Language Processing(NLP). WORD2HTN combines that paradigm and an algorithm to process it for generating task hierarchies. Our contributions are the following:

- We describe WORD2HTN, a new semantic learning algorithm for identifying tasks and goals based on semantic groupings automatically elicited from plan traces. WORD2HTN takes as input the execution traces as sentences with the planning components of the trace (operators, atoms and objects) as words . It then uses the popular semantic text analysis tool WORD2VECTOR [Mikolov *et al.*, 2013], which uses a neural network to learn distributed vector representations for the words in the plan trace. We discuss the WORD2VECTOR tool in Section 4 and give an intuitive explanation for why it learns good representations. To put it succinctly, distributed vector representations are learned based on the co-occurrence of the words in the input traces. From the vector representations, we can generate groupings through agglomerative hierarchical clustering. This groups words that are closer together, and repeats this process to build a hierarchy.

- We describe a new algorithm that extracts HTNs from the clustering. The results of the clustering can also be used to group goals and identify tasks. The similarity of the learned vector representations with the current and goal states (input to the planner) will be used as the method to decompose a higher level task in an HTN to the next action to be taken.

- We report and discuss our implementation of the WORD2HTN approach and demonstrate the progress and promise via examples in a logistics planning domain.

2 Semantic Word Embeddings and Vector Representations

For semantic reasoning about goals and tasks, we took inspiration from Natural Language Processing(NLP) techniques, specifically Word Embeddings. They are a way of representing, or embedding a word in an N-dimensional space as distributed vectors [Mikolov *et al.*, 2013]. This technique for learning word embeddings is popularly called WORD2VECTOR. The number of dimensions N can be chosen by heuristics or experimentally (we will discuss more on this in Section 4). To illustrate how words might look, let us consider a component of a domain $Location1$ in a 5-dimensional space. It maybe represented by $[0.5, 1.2, -0.6, 0.8, 0.4]$, The component $Location2$ might have a representation of $[0.4, 1.4, 0.5, 0.9, 0.3]$. Notice how all dimensions but one are close in value. This is a simplified example to illustrate how related components could be encoded in the vector model. Most of the dimensions would be similar when $Location1$ and $Location2$ are used frequently in similar contexts. Their vector representations would be learned by adjusting the dimensions from their initial random values in the same directions. It is important to note that we do not know what each dimension means with the WORD2VECTOR approach to learning word embeddings. In NLP, they are referred to as "latent semantic dimensions" because they implicitly encode properties, but what each dimension means is hidden and affected by the training process. See Figure 1 for an example of distributed vector representations for some components of a logistics (transportation) domain

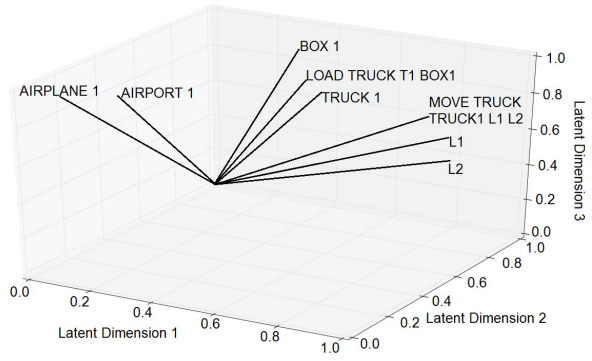

Figure 1: Sample Distributed Vector Representations of Components from Logistics Domain

Mikolov *et al.* showed how quality vector representations of words from a corpus (group of documents) can be learned using neural networks. The neural network starts with randomized initial weights, and corrects them with the training sentences (plan traces in our case) . The representations are learned by looking at words within a specified context window size. The context window size (C) defines the number of words (planning components) before and after a target word that is used to define the context. The target component's representation is adjusted to be closer to the words in it's context. When all the sentences (plan traces) are iterated over, the final N-dimensional representation (N is an input parameter) of each word in the vocabulary is learned. The distributed representations are learned for each word to maximize each word's similarity with the words that appear in its contexts. Additionally, words that do not occur within the same contexts would have differently aligned vectors. More details on this process is discussed in Section 4.

3 Definitions and Terminology

We use standard definitions from the planning literature, such as objects, atoms, actions, and state variables [Ghallab *et al.*, 2004] [Chapter 2]. We will refer to these constructs as *planning components*. They correspond to the "words" in the plan traces. We summarize our definitions for planning components below.

An *action* is a primitive (smallest) task with arguments that are constant components (e.g., load-truck truck1,location1). Note that the entire action component including the arguments is treated as one word in the plan traces while learning. An action can change the state of the domain when it is applied. Each action has a precondition and an effect. The precondition and effect of an action is a set of atoms. For example, "Move Truck1 Location1 Location2", is an action that has the precondition "Truck1 in Location 1". The effect can be "Truck1 in Location2".

An *atom* is a positive literal, such as "Truck1 in Location2". All the literals in the initial work were atoms (positive literals) as opposed to negations. "Truck1 not in Location2" is instead covered by "Truck1 in Location3" and the like.

A *state* is a collection of literals, indicating the current values for properties of the objects in a 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 by a series of preconditions, actions, and their effects. As an example, the plan trace to transport package1 from Location1 to Location2 could be as follows:

Precondition ("package1", "package1 in Location1", "Location1", "Truck1", "Truck1 in Location 1", "Location 1")
Action("Load package1 Truck1 Location1")
Effect("package1", "package1 in Truck1", "Truck1",
"Truck1", "Truck1 in Location 1", "Location 1")

Precondition("Truck1", "Truck1 in Location 1", "Location 1")
Action("Move Truck1 Location1 Location2")
Effect("Truck1", "Truck1 in Location 2", "Location 2")

Precondition ("package1", "package1 in Truck1", "Truck1", "Truck1", "Truck1 in Location 2", "Location 2")
Action("Unload package1 Truck1 Location2")
Effect("package1", "package1 in Location2", "Location2", "Truck1", "Truck1 in Location 2", "Location 2")

In the previous example, the goal only consists of a single atom, namely "Package1 in Location2". If the goal also requires that *Truck 1* be in *Location2*, then the goal would be the conjunction of two atoms "Package1 in Location2" and "Truck1 in Location2". Note that in the actual plan traces, that were fed into our algorithm, there was no separation of components into groups of actions, preconditions, and effects. These were added in the example for readability.

Lastly, a *task* denotes an activity that needs to be performed. Frequently to convert a goal into a task we encapsulate the goal with a task *accomplishGoal* that performs the actions needed from an initial state to achieve the goal. WORD2HTN will identify subgoals for achieving a goal. Each of these subgoals will be encapsulated in smaller subtasks thereby generating a hierarchical structure.

4 Learning Distributed Vector Representations for Plan components

We observe that distributed vector representations can be learned well for components in plan traces. In this paradigm, the natural-language text words are atoms, actions, and the objects of the domain (like Truck1). Similar components' learned vector representations are closer together. We used the WORD2VECTOR approach for generating word embeddings. Each sentence (plan trace) is separately processed from every other sentence in WORD2VECTOR. A sentence could either be a complete plan trace from a start state to a goal state, or even just a single action with its precondition and effects.

In our initial experiments, we chose the first option of complete plan traces (i.e., sequences of one or more actions) in a sentence. We chose this approach since the actions to reach the goal state are sequentially executed with the next action's preconditions requiring the previous action's effects. If the actions in the traces were ordered with no semantic reason, then it would help the learning by breaking the plan trace into sentences of one action per sentence. This would prevent inferring any incorrect relationships between components that are only coincidentally next to each other.

The WORD2VECTOR approach for learning the distributed representations, uses a shallow neural network model and model parameters that define how WORD2VECTOR learns the representations. The most important model parameters are: the number of dimensions N, the context window size C, and the learning rate α. After setting these, we pass in the sentences of the data set. Every component is encoded using one-hot encoding. This means every input component is a long vector of size V, which is the total number of components in the data set. Only one dimension of the vector in one-hot encoding is set to "1", and it is unique for each word. This is a sparse vector, which matters since it makes the multiplication operations on it cheaper. This sparse vector is the input form passed into the neural network. The edge weights of the neural network are randomly initialized. With every data point, the edge weights are adjusted by a learning rate (α) such that the target word and those within it's context have closer vector representations.

There are two common neural network structures in WORD2VECTOR .One is Continuous-Bag-of-Words (CBOW) and the other is Skip-Gram (SG). These are two ways of comparing a word to it's context, and that determines the network structure (see Figure 2; cf. [Rong, 2014] used with author's permission).

- **CBOW**. In CBOW, the input consists of all the words within the context window of the target word. In figure 2, the left side is the CBOW structure. The inputs are the context words for x_k, and these are $x_{1k}, x_{2k}...x_{Ck}$, where C is the size of the context window. Then the hidden layer's values are calculated from each of the input words multiplied (matrix multiplication) by the edge weights W_{VxN}, and then averaged. Note that V is the size of the vocabulary or the total number of plan components, and N is the number of dimensions of the learned vector representations.

  The hidden layer consists of nodes $h_1, h_2, ...h_N$ for a total of N nodes. The output value is the hidden layer value multiplied by the output edge weights W'_{NxV} . The output is again a V-dimensional vector like the input. It is expected to match the target word's one-hot encoding. The error is back propagated through the network and the edge weights are adjusted by the learning rate α.
- **SG**. In Skip-Gram, the network is inverted (refer to right side of figure 2). The input is the single target word x_k, and the output is compared to the each of the words that makes up the 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 just as in CBOW.

In the neural network, each component's one-hot encoding is being projected 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 along one of the N dimensions. The input edge weights are of dimensions 5x3. Each column is the representation of a basis vector of the N-dimensional space in the one-hot encoding form of the input. For example, the first latent 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 $Truck1$ would 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 $Truck1$ in the 3 dimensional space.

In CBOW network, the hidden layer's values would be the average of the projections of each input words. On the output

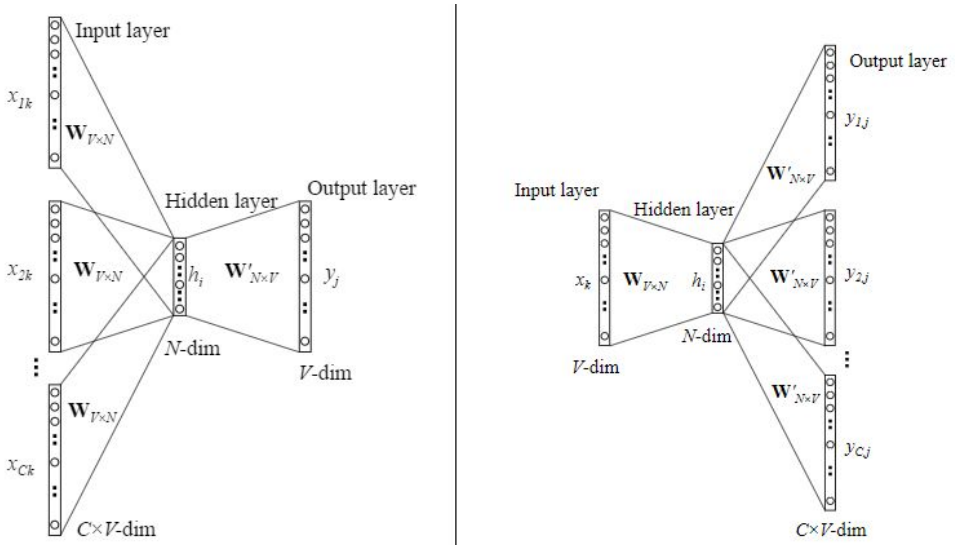

Figure 2: Comparison of CBOW and SG Neural Network Configurations for WORD2VECTOR.

side, the edge weights represent the N-dimensional representation of each word of the vocabulary. Column 1 of the output weights would be the vector representation for word 1 of the vocabulary. In the current 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 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] when word 2 and 3 are the most similar. Finally, we run a softmax function on the output to get the probability of each output word. The difference (error) with the actual result in the plan trace is then back propagated to update the edge weights.For more details on the math and error propagation, please refer to the paper by Rong (2014).

Once the vector representations are learned, we can analyze them for relationships. The similarity between two components is defined by the cosine distance of their vector representations. This is the cosine of the angle between the vectors. To get an intuitive idea see Figure 1 again, which shows how similar vectors can be more closely aligned. The similarity is calculated by taking the dot product of the normalized vectors of the components. Cosine distance is a standard metric for similarity between word embeddings in NLP. We have adopted it as well, and initial experiments have show good results with this metric.

5 Learning Goal and Task Hierarchies

The motivation for using vector representations is to transform the problem into a form in which we can apply more general and powerful algorithms. We transform it to a distributed vector representation using the WORD2VECTOR tool. This representation can then be used to apply clustering algorithms like Agglomerative clustering or K-means on the vector representations. We chose to use agglomerative clustering (bottom-up hierarchical clustering). This will generate a hierarchical grouping of all the actions by their semantic similarity (cosine distance).

We generate plan traces for the logistics domain by hand coded HTNs. It must be noted that in addition to the preconditions and effects for each action, we also insert the atoms associated with the objects in the action. This means that "Location1 in City1" will be added because it is an atom associated with "Location1" which is an object in the action "Load package1 Truck1 Location1". Related atoms are added to help learn all relationships in the domain. Another important point is that the order of atoms, and the order of objects around those atoms is randomized for each trace. Only the fact that the preconditions occur before the action, and the effects occur afterwards is fixed. For example:

Precondition ("package1", "package1 in Location1", "Location1", "Truck1", "Truck1 in Location 1", "Location 1", "Location 1", "Location1 in City1", "City 1")
Action ("Load package1 Truck1 Location1")
Effect ("package1", "package1 in Truck1", "Truck1",
"Truck1", "Truck1 in Location 1", "Location 1", "Location 1", "Location1 in City1", "City 1")

Once the component's vector representations are learned, we then use it to identify clusters of atoms and actions for grouping. One of the groupings that we can learn are the sets of actions which are semantically related, and make tasks.

After identifying a semantically related set of actions, we can identify the atoms achieved (net effects), and the initial atoms needed to start it (net preconditions) using knowledge of each action's preconditions and effects. We thus identify the pair *(net preconditions, net effects)* as a task, which is used to build an HTN by combining it with other action groups that

have related preconditions or effects. Thus we can learn tasks from the grouping of planning components by their similarity (cosine distance).

Groups are combined hierarchically based on the strength of the similarity between them (cosine distance). An action is linked to another one when there are atoms that are common to both actions. This can happen, for example, when the effects of one action are in the preconditions of another one.

6 Implementation

6.1 Experimental Domain

In order to test the methodology, we use a slight variant of the logistics transportation domain [Veloso, 1992]. In this domain packages must be relocated between locations. These locations can be in the same city or in a different city. Trucks are used to transport packages between locations in the same city. Airplanes are used to transport packages between airports (i.e., a special type of location) in different cities. In our variant, we use planets to add another transport layer. To travel between planets rockets are used. Rockets can travel between space stations. In our experimental setup, we use state configurations consisting of two planets, cities in planets, and locations within cities. Plan traces for transporting the package from a randomly-selected location in Planet1 to a location in Planet2 were generated. These plan traces were then given into WORD2VECTOR to learn the vector representations. The plan traces consisted of sequences of operators. Each operator was preceded by its preconditions and succeeded by its effects. In addition to the precondition and effects, the atoms associated with the objects that the operator affects were included, such as "Location1 in City1", even though it is not explicitly a precondition. By including all the object information associated with the objects in the operator, we can better learn relationships and hierarchies in the domain. One example of learning such a hierarchy is grouping all the actions associated to a city in one task as in figure 3.

6.2 WORD2HTN Setup

To train the model, we tune parameters of the WORD2VECTORimplementation for it to learn quality vector representations. The important parameters that were tuned are the following:

Similarity Evaluation. The similarity between two components is the dot product of their normalized vectors. This is also equal to the cosine of the angle between the components' vectors. The idea is that if the principal components of the two vectors are both high (closer to 1), they will contribute more to the similarity value. On the other hand if either one is small, then it will contribute less to the similarity. Since these are unit vectors, the sum of all the dimensional products can be at most 1 (when they are identical).

The similarity cutoff of 0.60 was chosen based on the experiences of other engineers and researchers in NLP who have used WORD2VECTOR. Our initial experimental results show this value to be good as well. We are still searching for formulas or metrics to help choose a good cutoff, or to dynamically change the cutoff when grouping more dissimilar components into hierarchically larger groups.

If redundant preconditions are consistently present in the training data, then the association with the action will be learned. However, if atoms and objects appear infrequently, and with other unrelated actions as well, then the association is very weak. It is important to note that WORD2VECTORonly learns similarities by frequent co-occurrences. We cannot learn an action's cause and effect. What we can learn is statistically significant grouping of actions into groups (tasks) with subgoals based on their inferred semantic relationships.

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. Fewer dimensions would also cause the similarity between vectors to be higher than it ought to be. It is better to err on the side of more dimensions, even if the training time increases. As a rule of thumb, we set the dimension size to twice the number of objects in the domain. The intuition was that every atom and action uses a subset of the objects in the domain. Therefore, setting the dimension size as a function of the number of atoms or objects would give enough space to distribute the vector representations for effectively capturing the relationships. This is not a hard rule, but an intuition that we used. In our experiments, there were 26 objects, and so we used 52 dimensions.

Another approach to finding an appropriate value for any parameter would be to vary one parameter for multiple trials and compare the results. So increase the number of dimensions, and compare the results with expected similarities. When the similarity of two components that was expected to be high, shows little change with increasing dimensions, then stop increasing the number of dimensions. If we graph the dimension size on the x-axis, and the similarity of the test components on the y-axis, the point would look like the knee of the graph. It is the point when the incremental improvement with parameter changes is minimal. Needless to say, this approach requires multiple iterations, and having a set of plan components for which we expect high similarity. Although this approach can be time consuming, it can help train a good model.

Context Window Size. This is the size of the window (or number of words) around the target word which defines its context. The context window size ought to be large enough to cover the relevant words. When choosing a window size it is better to err on the smaller side, and use more training data. For our experiments we chose a window size of 10. A smaller window size ensures that the atoms and objects that fall inside the context are actually relevant to the target word. Recall that every sentence is a plan trace consisting of a series of actions, atoms, and objects. To ensure that as many of the relevant planning components are associated with the target action, it also helps to randomize the order of the atoms and objects in the preconditions and effects. Randomizing the order ensures that the many of the relevant atoms definitely fall

into the context window (especially if it is small) in at least some of the training data. However, if the window size is too large, it is more likely to pick up noise or artifacts from the training data that are not semantically relevant. An exaggerated example to illustrate this problem would be that the window size is larger than the entire plan trace. In that case, every word's context is every other word in the trace and that would be incorrect. If a sentence only contains one action with preconditions and effects, then setting the window size to cover the entire sentence makes sense, since the context is the entire sentence. This is however not the case, and a plan trace typically contains many actions.

6.3 Ordering actions and Hierarchical Clustering.

After the vector representations of the model are trained, we can then begin clustering the actions, atoms, and objects into groups by their semantic similarity. One effective way to do this is agglomerative clustering. This is a bottom-up hierarchical clustering approach. What that means is, in every iteration the closest components are grouped together into a new unit, and the process is repeated. The distance metric is the cosine distance between the vectors. The smallest angle between any one of a group of vectors and the new vector to be added is the metric for deciding the next group to make. For example, if the action "Move Truck Location1 Location2" is closer(more similar) to the atom "Truck in Location2" (as per its vector representation) than any other pair of vectors, then those two words would be grouped first before others. Later the new group maybe combined with "Truck in Location1" to form another group. In this manner, we can group semantically related components together. An example that shows actions being grouped hierarchically is shown in figure 3. Other atoms and objects in the graph were removed for visibility as the graph would be cluttered with them.

After the clustering is complete, we have different subtasks at different levels of the hierarchy. When planning and solving, in order to decide what action to take we use the HTNs and vector values learned by WORD2VECTOR and search with the input current state's atoms and objects, as well as the goal state's atoms and objects (like "Package1 in Location5", "Package1", and "Location5"). The task that has the highest similarity with the current state and goal state, will be chosen. Then that task will be decomposed further to choose the subtask with the highest similarity, and so forth until a primitive action is reached that can no longer be decomposed. This smallest action is then executed to approach the goal state. Please note that we have not yet completed the code to choose the action and solve new problems from the learned HTNs. We have finished the agglomerative clustering from the learned vector representations, an example of which is shown in figure 3.

6.4 Initial Experiments and Results

The training plan traces were generated from hand coded rules on the logistics domain that was specified. There were 9 different plan traces because of 9 different starting states. All traces had the same goal of delivering $Package1$ to a location in $Planet2$ from a location in $Planet1$. The average length of the plan traces was 221 words (actions, atoms, and objects). The traces were fed into an implementation of WORD2VECTOR in python [Řehůřek and Sojka, 2010]. The implementation iterated over the different plan traces 1000 times to help the vector representations stabilize. Multiple iterations are typical in Natural Language Processing as the learning rate (α) is typically a small value like 0.001 (the value we used). Finally, after the model was trained, the resulting vectors were put through an agglomerative clustering function (as described in the previous section). This results in hierarchical grouping of the planning components. We can see some of the grouped actions in Figure 3. Other components (atoms and objects) were removed for visibility.

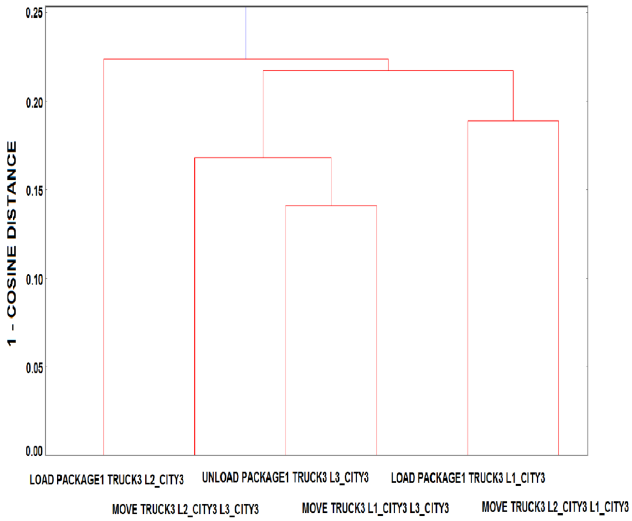

Figure 3: Agglomerative Hierarchical Clustering of Actions in One City

The results met our expectations. The hierarchy learned from the plan traces, was as we constructed the domain. 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) was in a group. Those two groups were combined together first, before being combined with the actions to transport the package across planets (by rocket).

While the hierarchy meets our expectations, we have yet to implement the code to use the learned HTNs and vector representations to solve problems in the domain. We would also like to analyze WORD2HTN performance in a domain that does not have a clear hierarchy.

7 Related Work

In order to understand how our work relates or compares to other work in the field, let us look at some of the planning and HTN literature. Most existing work on learning HTN planning knowledge focuses on using symbolic techniques to group the actions. For example, ICARUS [Choi and Langley, 2005] learns HTN methods by using skills (i.e., abstract definitions of semantics of complex actions). 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 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.

Another example is HTN-Maker [Hogg *et al.*, 2008]. HTN-Maker uses task semantics defined as (preconditions,effects) pairs to identify segments (i.e., sequences contiguous actions) in the input plan trace where the (preconditions, effects) are met. Task hierarchies are learned when a segment is identified as achieving a task and the segment is a subsegment of another larger segment achieving another task. This includes the special case when the subsegment and the segment achieve the same task. In such a situation recursion is learned.

The only other work that we know which learns HTN planning knowledge using training data to build a model is HTNLearn [Zhuo *et al.*, 2014]. HTNLearn transforms the input traces into a constraint satisfaction problem. Like HTN-Maker, it assumes (preconditions,effects) 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 subtask 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 similarities and groups are elicited from the learned vector space representations.

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* [Parr, 1998; Diettrich, 2000]. 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. In our work WORD2HTN the hierarchical abstractions are learned based on the co-occurrence of the objects,atoms, and actions in the plan traces.

8 Conclusions

We have described WORD2HTN, a new algorithm for learning hierarchical tasks and goals from plan traces in planning domains. WORD2HTN first learns semantic relationships from plan traces and encodes them in distributed vector representations using WORD2VECTOR. Then this vector model is used to combine the components that are most similar into hierarchical groups using agglomerative clustering. This hierarchical grouping is used to define tasks and build HTNs.

The next step in our research is to implement the code that uses the learned HTNs and vector representations to choose actions for reaching the input goal states. We also want to find an appropriate metric to measure the efficiency of the plan trace produced by the planner using the learned model. After completing our experiments in the logistics domain, we will apply this approach to different domains such as Blocks World to test its validity. We will also conduct experiments with several benchmark planning domains from the automated planning literature.

Thus far the experiments that we have done have been with deterministic traces. However, the reason we started this research, is because we think this method can work well with non-deterministic traces as well. Moving forward, we will generalize our approach to probabilistic actions with multiple outcomes. We plan to generate plan traces for learning from the probabilistic actions, where each plan trace will result from a probabilistic execution of of actions. We believe that the rest of the WORD2HTN algorithms will still work because the components with shared contexts will still have similar vector dimensions. One difference is that the similarity of the effects of an action would depend on the frequency of occurrence (probability distribution) of the different 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 learned similarity of that atom with the action to load the truck as well (albeit a weaker similarity). We will investigate this hypothesis both theoretically and experimentally with non-deterministic plan traces.

Acknowledgments

This research was funded in part by 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

[Choi and Langley, 2005] Dongkyu Choi and Pat Langley. Learning teleoreactive logic programs from problem solving. In *Inductive Logic Programming*, pages 51–68. Springer, 2005.

[Diettrich, 2000] Thomas G Diettrich. Hierarchical reinforcement learning with the maxq value function decomposition. *J. Artif. Intell. Res.(JAIR)*, 13:227–303, 2000.

[Erol *et al.*, 1994] Kutluhan Erol, James Hendler, and Dana S Nau. Htn planning: Complexity and expressivity. In *AAAI*, volume 94, pages 1123–1128, 1994.

[Ghallab *et al.*, 2004] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning: theory & practice*. Elsevier, 2004.

[Hogg *et al.*, 2008] Chad Hogg, Héctor Munoz-Avila, and Ugur Kuter. Htn-maker: Learning htns with minimal ad-

ditional knowledge engineering required. In *AAAI*, pages 950–956, 2008.

[Mikolov *et al.*, 2013] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Workshop proceedings of the First International Conference on Learning Representations*, Scottsdale, Arizona, May 2013. International Conference on Learning Representations.

[Nau *et al.*, 2005] Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, Dan Wu, Fusun Yaman, Héctor Muñoz-Avila, and J William Murdock. Applications of shop and shop2. *Intelligent Systems, IEEE*, 20(2):34–41, 2005.

[Parr, 1998] R. Parr. *Hierarchical Control and learning for Markov decision processes*. PhD thesis, Univ. of California at Berkeley, 1998.

[Řehůřek and Sojka, 2010] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[Rong, 2014] Xin Rong. word2vec parameter learning explained, 2014.

[Veloso, 1992] Manuela M Veloso. Learning by analogical reasoning in general problem solving. Technical report, DTIC Document, 1992.

[Zhuo *et al.*, 2014] Hankz Hankui Zhuo, Héctor Muñoz-Avila, and Qiang Yang. Learning hierarchical task network domains from partially observed plan traces. *Artificial intelligence*, 212:134–157, 2014.