2017

Learning Hierarchical Task Networks Using Semantic Word Embeddings

Sriram Gopalakrishnan
Lehigh University

Learning Hierarchical Task Networks Using Semantic Word Embeddings

by

Sriram Gopalakrishnan

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

May 2017

The thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

_____

Date

_____

Thesis Advisor

Dr. Héctor Muñoz-Avila

_____

Chairperson of Department

Dr. Daniel P.Lopresti

Acknowledgements

A big thank you to Dr. Héctor Muñoz-Avila, my thesis advisor and guide through the research process. An additional thank you for your research jokes and stories, they helped to make my tasks lighter and enjoyable. To Dr.Ugur Kuter, your perspective and guidance made a huge impact to my work. Thank you so much for taking the time amidst your demanding schedule to collaborate. Lastly, to Noah Reifsnyder, and Sam Nguyen, my awesome lab mates and research collaborators. Your work ethic, and dedication helped push this work further. I expect great things from you two in the future.

CONTENTS

TABLE OF FIGURES

Abstract:

This thesis describes WORD2HTN, which is a novel and semantic approach for learning hierarchical task networks (HTN) and semantic division of goals from input plan traces. The semantic relationships are learned using machine learning to get the vector representations of the components of the plan trace. The semantic relationships are used to learn hierarchical landmarks, which in turn are used to make semantically divided HTNs. These learned HTNs can then be used for subsequent new problems in the domain that have a similar structure with the problems in the input plan traces. This work also improves the learning algorithm to include arithmetic conditions and effects. WORD2HTN was tested on 3 deterministic domains. These are Logistics or Transportation domain, Abstract Graph domain, and the Malmo interface for the Minecraft game. We show that WORD2HTN learns semantically divided HTNs. We also experimentally demonstrate that HTN planners using this have an exponential speedup in information-dense domains over the state of the art classical planner. Finally, we show that the HTNs learned in Minecraft can be used to achieve tasks faster with a cooperative agent controlled by the HTN planner's output.

Chapter 1: Introduction

Automated Planning is a field of research that involves the solving of problems in a domain by generating plans to transform the state of the world to satisfy the goal conditions. There are specific terms and concepts common in planning that is described in the detail in the next section on the preliminaries. One of the general components is the planner. A planner is the program or algorithm that generates the plan (sequence of actions) to satisfy the goals. The planner has access to the rules or models of the domain, which specific how states can be changed. A planner that has access to only the action models, which specify what actions (low-level) can be taken in the domain, would have to search across a typically large space of possible states to reach the goal states. This is a brute-force approach, and becomes exponentially slower as the size of the domain increases.

Classical planners are those that only use the low-level actions (called operators) of the domain, and relationships between objects and states to heuristically search the problem space and approach the goal state. In contrast, Hierarchical Task Network (HTN) planners use a library of methods to inform its search for plan to reach the goal state.

In many practical planning domains, a planner is typically exposed to many potentially feasible planning trajectories it could follow to achieve some desired goals. It is reasonable to expect automated planners to have access to previous logs of trajectories followed by domain experts when solving problems in that planning space; e.g., logs of a

remotely-controlled UAV might repeat a flight corridor used by controllers even though there could be multiple other trajectories that could be followed. Yet, even if it is available, most state-of-the-art classical planners [Fox and Long, 2003] planners do not use this information to guide their search because the planner does not know why and how this information is relevant to a planning problem. The traditional approach to help automated planners cope with such large amounts of planning spaces is to have human domain experts encode the necessary information into the planning problem itself, and the domain definitions (i.e. action definitions). This can present a huge knowledge-engineering burden. It also requires the human experts to be able to know how to encode it, and thus some knowledge of the internals of the planner. This approach is not feasible as the complexity of problems and planners increase.

This thesis presents a new HTN (Hierarchical Task Network) learning system that addresses this challenge through a new hierarchical landmark learning method. The learning algorithm identifies subgoals in the input plan traces using semantic-based clustering of the parts of the plan trace. Specifically, WORD2HTN takes or interprets plan traces as a sequence of words. Each word can be an action or atom (positive literal). The algorithm initially treats these plan traces as a set of sentences and uses semantic text analysis to find clusters of words (actions and atoms) based on semantic similarities. The output is a hierarchical goal structure. From the output, we construct HTN methods. The HTN methods also has the semantics (i.e., preconditions and effects) on tasks and the decompositions that accomplish those goals. This thesis details the formalisms used for the WORD2HTN approach and the learning algorithm. The formalism section is followed by the evaluation of the algorithm. Our experiments demonstrated that

WORD2HTN can capture the relevant semantic knowledge for task decomposition, and learn semantically informed HTNs. In our first experiment, we observed that WORD2HTN tends to learn balanced HTNs. By this we mean, the tree representation that captures the task decomposition is balanced as opposed to right-recursive or one-sided. In our second experiment, we compared planning performance with the learned HTNs, with that of the state-of-the-art classical planner Fast-Downward [Helmert, 2006; Helmert et al., 2011]. We used J-SHOP, a Java-implementation of the well-known HTN planner SHOP [Nau et al., 1999]. To compare appropriately, we used three heuristic/search settings in the Fast-Downward planner that used landmark extraction for efficient search. Our results demonstrated that WORD2HTN can learn the semantic division of tasks using landmarks and generate HTNs with which the average runtime for planning in information-dense domains is exponentially reduced as compared to Fast-Downward.

Chapter 2: Preliminaries

We will first introduce the terminology that will be used throughout this thesis, as well as the fundamental concepts that this work will build upon.

2.1 Atom:

One of the fundamental terms in planning is the concept of an atom. An atom is a positive literal (not a negation) in first order logic. A simple example is "*The truck is at location_1*". In this example "*truck*" is the object, the property is "*at",* and *location_1* is the value of the property. An atom can be defined with variables, in which case it is a lifted atom as opposed to a grounded atom. For example, if we don't know which truck, or which location, then the values are variable and so is a lifted atom. If we do know the specific values, like "*The Red Truck*", and "*The gas station location*", then it is a Grounded Atom. Typically, when we reference an atom, it will not be in plain English, but in an abbreviated form of the type "<*object*> <*property*> <*value*>". The equivalent concrete example would be "*Truck_1 at location_1*". If the values are variable then it would be of the form "*?Truck at ?location*". The "?" denotes a variable.

2.2 States, Operators and Actions:

A *State S* is a set of grounded atoms. An *Action* is formalized as the triple *(h, p, e)*. *h* is the header of the action structure, and in turn stores the name of the action and the list of objects referred to in the action. *p* is the set of precondition atoms. These are the atoms that must be true in the current state for the action to be applicable, i.e. $p \subseteq S$. *e* is

the set of effects, which can be added atoms, as well as deleted atoms. The effects are represented as lists of *add(atom)* and *delete(atom)* statements. The result of applying the action a to a state *S*, is a new State *S'* that is:

$$S' = (S\text{-deletedAtoms}) \cup \text{addedAtoms}$$

2.3 Planning problem:

A Planning problem P is defined as the triple (A,S_0,G) where A is a set of actions, S_0 is the set of atoms that define the starting state, and G is the set of atoms that need to be true in the goal state.

2.4 Plan (action plan):

A Plan π consists of a sequence of actions "$a_1, a_2, a_3 \ldots a_n$". π solves a problem P if the following statements are true.

(1) The first action a_1 can be executed in the initial state S_0, i.e. the preconditions are satisfied in S_0. The subsequent actions "$a_2, a_3 \ldots a_n$" are applicable in the states produced by the previous actions. So a_2 must be applicable in S_1, which is the state that results from applying action a_1 to state S_0.

(2) The state S_n that results after applying the last action a_n from π, should contain the atoms of the Goal set *G*. So we generate S_n from S_0 by applying π

2.5 Plan Trajectory and Plan Traces:

We represent the state and actions of a plan π applied to an initial state S_0, as a plan trajectory. It is a sequence as follows: "$S_0,a_1, S_1,a_2, S_1,a_2, \ldots S_n,a_n$".

For this work WORD2HTN, the Plan Traces that are given as input to the algorithm, is defined as follows. It is the sequence of action with each action preceded by its preconditions, and succeeded by its effects. A plan trace would be of the form

"$p_{11}, p_{12}, a_1, e_{11}, e_{12}, \ p_{21}, p_{22}, a_2, e_{21}, e_{22}, \ldots \ p_{n1}, p_{n2}, a_n, e_{n1}, e_{n2},$"

Where p_{n1} and p_{n2} would be the first and second preconditions of the n^{th} action. Similarly, e_{n1} and e_{n2} would be the first and second effects of the n^{th} action. There can be less or more than 2 preconditions or effects, it is not that only 2 are allowed in the plan traces.

2.6 HTN Formalism:

An HTN planner takes a set of *Tasks* to achieve. These tasks can be primitive or compound. A primitive task is just the headings of the action(s) with the associated list of parameters that define the action (grounded). Recall that a specific action is a grounded operator.

A compound task has a name, and a list of parameters (objects, values). These define the compound task (as in the English sense of the word), to be performed and requires more than one sub task to complete. The subtasks can be compound or primitive tasks.

A *method m* is defined by a triple *(h p T). h* is the heading of the method, which is a compound task. *p* is the collection of preconditions that define the starting state of the method. Finally, *T* is the ordered list of sub tasks (primitive or compound) that is required to achieve the task in the heading.

2.7 HTN planning problem:

A planning problem with HTNs, is represented as a tuple of 4 elements, $H = (S_0,T,M,A)$. S_0 is the set of atoms defining the starting state. T is the ordered set of Tasks to achieve. M is the set of methods that can be used to achieve the tasks in T. Lastly, A is the set of actions that can be executed in the domain.

2.8 Planning and Solving an HTN problem:

Each of the tasks t in T, is achieved in one of two ways:

(1) If t is a compound task, then we find a method m whose header matches the task, and can be applied in the current state. The task t is replaced by the subtask ordered list from m. These represent the sub tasks that need to be accomplished for the main task t (that was replaced) to be achieved.

(2) If t was a primitive task, and there is an action a that is applicable in the current state, then t is removed from T, and the action a is added to the solution plan π. Note that π starts out empty.

We always decompose the first task in the list until we reach actions that we can execute. As we reach actions to execute, they are added to the plan π. A solution plan π for the planning problem H (as previously defined), is one of the possible plan(s) produced using the aforementioned 2-step recursive process. We formally state this as "π *achieves T from* S_0".

2.9 Annotated Task:

An annotated task is defined as a triple $t= (\tau,p,g)$. τ is a compound task. p and g are the tasks preconditions and goals respectively. For this thesis, we use the notation $t.g$ to refer to the task's goals. $t.p$ refers to the task's preconditions. The way we define annotated tasks, gives the framework for defining compound tasks. To put it together, when a plan π achieves task t from a starting state S_0 such that $t.p \subseteq S_0$, then any plan π produced by the methods in M (learned by *WORD2HTN*), results in the state S_n such that $t.g \subseteq S_n$. S_n is the state of the world after applying all the actions of π in sequence from starting state S_0.

In this thesis, we are actually learning the Tasks and Methods from a set of plan traces and action definitions, given as the pair (Π , A). The result of the algorithm *WORD2HTN* is a set of annotated tasks and methods (T,M). The task set will include primitive tasks, which are achieved with a single action, as opposed to a compound task. The solution or result from *WORD2HTN* of (T,M) is correct if: given an annotated task $t=(\tau,p,g)$, and any starting state S_0 such that $t.p \subseteq S_0$, the result is $t.g \subseteq S_n$. S_n is the state of the world after applying all the actions of the plan generated (π) in sequence from starting state S_0.

2.10 Additional Concepts, Standards and Programs:

In automated planning there are two broad divisions of planners viz., classical planners and HTN planners. Classical planners only use operators (lowest level of actions), and the domain space to solve or achieve a given set of goals. HTN planners on the other hand, use a library of methods which specify task division, to solve the input tasks.

The current state of the art classical planner is the Fast Downward planner [Helmert et al., 2006]. It has different heuristics for exploring the domain space, which helps or guides the planning. It is the classical planner that we used for comparisons

JSHOP is a java implementation of the Simple Hierarchical Ordered Planner (SHOP) [Nau et al., 1999] which is an HTN planner. We have used it for the experiments run in this thesis.

Chapter 3: Word2Vector

A critical step in our WORD2HTN algorithm is learning embedded vector representations of the atoms and actions found in the input plan traces. This is needed for semantic reasoning about goals and tasks. We used a technique from Natural Language Processing (NLP), called Word Embeddings. These are vector representations of a word in an N -dimensional space. The words from the input text are learned as distributed vectors using *word2vector* [Mikolov et al., 2013]. They are distributed such that words which co-occur or have shared contexts have more similar representations, or are more aligned, than those that are unrelated. A simplified partial example is illustrated in Figure 1.

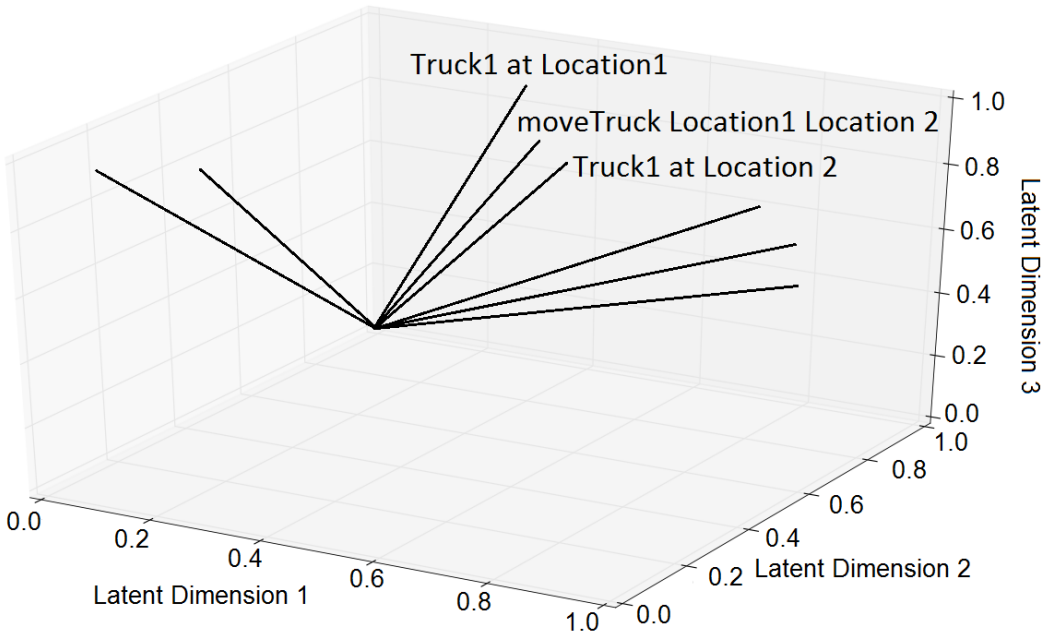

Figure 1: A good vector representation of 2 atoms and an action that are semantically related

The size of the dimensional space for which the vectors are learned, is represented as *N*, this is a parameter that can be tuned. It can be chosen using heuristics and/or experimentally. We will discuss this later in this section.

To illustrate how atoms and actions as words embeddings might look, let us consider the atom "*Box1 in Location1*" in a 3-dimensional space. It may be represented by [0.5; 0.2; 0.6]. The symbol "*Box1 in Location2*" might have a representation of [0.4; 0.25; 0.1]. This is an over-simplified example to illustrate how related symbols could be encoded in the vector space model. In this simplified case, 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 towards each other by moving them towards the words in their context.

The dimensions of the vector representations have no defined meaning, such as *distance in the x direction*. Instead, they are *latent semantic dimensions.* This means that they implicitly encode some relationship or dimension of the domain. However, what each dimension eventually represents depends on the initial random positions of the word embeddings, and how the plan traces (input data) adjust the representations.

3.1 Workings of Word2Vector:

We need to understand how *Word2Vector* works to understand the type of vector representations learned, and why it helps for learning HTNs

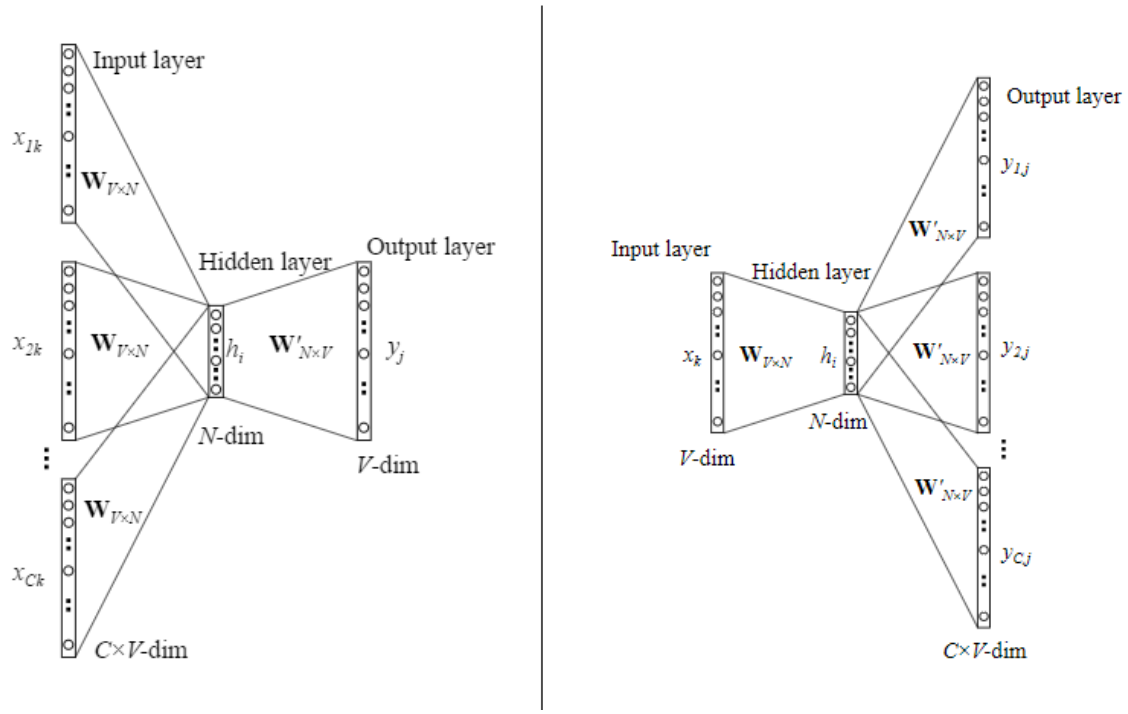

Figure 2: Comparison of CBOW and SG Neural Network Configurations for WORD2VECTOR. Figure from [Rong, 2014].

The central idea of *Word2Vector* is that words which occur close to each other, or share common words in their contexts, are more similar. More similar words will have more similar vector representations (we will define similarity later in this section). The input to *Word2Vector* is a collection of sentences, and each sentence represents a plan trace from a starting state of the world, to a state that satisfies a goal condition(s). Each sentence or plan trace is independently processed from every other sentence in *Word2Vector*. A sentence could be a complete plan trace from the start state to the goal state (as in our experiments), or it could even just be a single action with its precondition and effects. We found that the former approach gave better word embeddings, and attribute the improvement to the fact that there was more contextual information. We also chose to use

full plan traces as sentences since the actions to reach the goal state are often sequentially related by their preconditions and effects. If the actions in the traces were ordered with no semantic reasoning, then it would be incorrect to assume that actions occurring close to one another have some relationship. If that were the case, then it would make sense to break the plan trace into sentences of one action per sentence. This would prevent inferring any incorrect relationships between words that are only coincidentally in each other's context.

The core *Word2Vector* algorithm uses a shallow neural network model (see Figure 2) and model parameters that define how *Word2Vector* learns the representations. The most important model parameters are: the number of dimensions N (of the vector representations), the context window size C, and the learning rate α. After setting these, we pass in the sentences of the data set. Every element or word of the sentence is 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 entry/position set to 1 is unique and represents that word. This is the input form passed into the neural network. The edge weights of the neural network are randomly initialized. With every trial or run of the neural network during training, the neural net adjusts the edge weights by the specified learning rate α. The updates are done such that the target element 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 neural network structures in *Word2Vector*. One is Continuous-Bag-of-Words (CBOW) and the other is Skip-Gram (SG). These are two ways of relating a word to its context, and that determines the network structure (Figure 2).

3.2 Continuous Bag of Words (CBOW) and Skip-Gram (SG) network:

In CBOW, the input consists of all the words within the context window of the target word. So if the context window is of size 6, then the input will be the 6 words surrounding the target word (and define it's context). 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 word's representation multiplied (matrix multiplication) by the edge weights $W_{V\times N}$, and then averaged. V is the size of the vocabulary or the total number of symbols, and N is the number of dimensions (tunable) for each vector representation that is learned. Recall that the input is in one-hot encoding as described in the start of this section. The hidden layer consists of nodes h_1, h_2,…h_N for a total of N nodes corresponding to the N-dimensions of the vector representation.

The output value is the hidden layer value multiplied by the output edge weights $W_{N\times V}$. 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 or difference in the values is back propagated through the network and the edge weights are adjusted by the learning rate.

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 expected to be the words that makes up the

context in which the target word was found (y_{1k}, y_{2k},….,y_{Ck}). This network architecture tries to bring the single target word's vector representation closer to the words in its context. The difference or error is back propagated to updated the edge weights just as in CBOW.

3.3 Intuitive Understanding of CBOW and SG

In the neural network, each input element's *one-hot encoding* is being converted into an N-dimensional representation. 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 one of the dimensions in the N-dimensional representation of the input. The input edge weights are of dimensions VxN, which is 5×3 for our example. Think of each column as the representation of a hidden layer's basis vector in the one-hot encoding format of the input. For our example, let's say the first latent dimension's representation in the input 5-dimensional vector form would be [0.1 ,0.02 ,0.5, 0.3, 0.7]. Let us say that the input word is the atom "*Box1 in Location2"* and it's one-hot encoding 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 word embedding space. This is the projection of the input word onto each of the basis vectors. Therefore, the values of the hidden nodes h_1, h_2, h_3 in our example would be the projection of the input word "*Box1 in Location2"* in the 3 dimensional representation.

If the CBOW approach is adopted, then the hidden layer values would be the average of the projections of each input words. On the output side, the edge weights are a

NxV matrix. Each column represents the N-dimensional representation of each word of the vocabulary. So in the current example, the edge weights would be in a 3x5 matrix. 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]. Finally, we run a softmax function on the output to get the probability of each output word. The difference (error) with the actual output word(s) is calculated. The error is back propagated to update the edge weights. For more details on the math and error propagation, please refer to the paper by Rong[Rong, 2014].

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

Chapter 4: Word2HTN

4.1 Learning HTNs with Landmarks and Word2Vector

Algorithm 1 in Figure 4.1.1 shows a high-level description of WORD2HTN. For this section, we will describe the algorithm by referring to specific line numbers of this algorithm. The WORD2HTN algorithm uses two global variables: *AllMethods*, which keeps all the methods learned so far, and *processedSubsets* which holds the list of plan trace segments processed so far. The first step in WORD2HTN is to run word2Vector to generate the vector representations for the actions and atoms in all the plan traces (Line 4). Then we perform a Hierarchical Agglomerative Clustering (HAC) [Ward et al., 1963] on the vector representations and extract the landmarks (Line 5).

For our work, we ran HAC using the shortest cosine distance between the points of two groups as the clustering metric. HAC repeats the grouping or clustering of data points until we have one large group, and a hierarchy describing how the points were grouped. The output of the HAC process is a *clustering matrix* representing how the words were grouped hierarchically. From the HAC matrix, we can determine the landmarks for a specified set of words (the vocabulary of the traces). Given two clusters of points U and V, the landmark candidate u from U is defined as follows:

$$u = \arg\max_{u \in U} \sum_{v \in V} sim(u, v)$$

Similarly, we find the landmark candidate from V. Then we select from among the two candidates, the landmark with the higher score. To put it succinctly, a landmark u is the atom in the grouping of (U,V) that has the highest similarity with the atoms of the other

set. If there is an atom, that all plan traces go through, then this atom would be detected as a landmark.

```
1: global AllMethods <- ∅
2: global processedSubsets <- ∅
3: procedure WORD2HTN(Π):
4:   Embeddings <- RunWord2Vector(Π)
5:   L <- GetHACLandmarks(Π,Embeddings)
6:   for l ∈ L do:
7:     tracesSubset <- filter(Π,l)
8:     if tracesSubset ∉ processedSubsets then:
9:       processedSubsets.add(tracesSubset)
10:      ConstructMethods(tracesSubset,l)
11: procedure ConstructMethods(traces,l):
12:   leftTraces <- PreceedingSubtraces(traces,l)
13:   rightTraces<- succeedingSubstraces(traces,l)
14:   A <- GetHACLandmarks(leftTraces,Embeddings)
15:   B <- GetHACLandmarks(rightTraces,Embeddings)
16:   for a ∈ A do:
17:     tracesSubset <- filter(leftTraces,a)
18:     if tracesSubset ∉ processedSubsets then:
19:       processedSubsets.Add(tracesSubset)
```

```
20:        if CountActions(tracesSubset) <=1 then:
21:          M_A <- MakeMethod(tracesSubset)
22:        else:
23:          M_A <- ConstructMethod(tracesSubset,a)
24:    for b ∈ B do:
25:      tracesSubset <- filter(rightTraces,b)
26:      if tracesSubset ∉ processedSubsets then:
27:        processedSubsets.Add(tracesSubset)
28:        if CountActions(tracesSubset) <=1 then:
29:          M_B <- MakeMethod(tracesSubset)
30:        else:
31:          M_B <- ConstructMethod(tracesSubset,b)
32:    M_T <- MergeMethods(M_A,M_B)
33:    AllMethods <- AllMethods U M_T
34:    return M_T
```

Figure 3: Algorithm 1- The Word2HTN procedure that learns HTNs from Plan Traces

```
1: procedure MergeMethods(M_A,M_B):
2:    M_T <- ∅
3:    for m_a ∈ M_A do:
```

```
4:      for m_b ∈ M_B do:
5:        m_a'<- m_a
6:        m_b'<- m_b
7:        C <- commonObj(m_a[Task], m_b[Task])
8:        m_a'[Task.g] <- filter(m_a[Task],C)
9:        m_b'[Task.g] <- filter(m_b[Task],C)
10:       m_t[Task.g] <- m_a'[Task.g] U m_b'[Task.g]
11:       diff <- changes(m_a'[Task.g],m_b'[Task.g])
12:       m_t[Task.g] <- m_t[Task.g] – diff
13:       m_t[Pre] <- m_a[Pre] U m_b[Pre]
14:       m_t[Pre] <- m_t[Pre] – m_a'[Task.g]
15:       m_t[Pre] <- filterAtoms(m_t[Pre],C)
16:       m_t[ST] <- { m_a'[Task.g], m_b'[Task.g]}
17:       M_T <- M_T U {m_t}
18:       AllMethods <- AllMethods U {m_a'}
19:       AllMethods <- AllMethods U {m_b'}
20:     if |M_A|= 0 then:
21:       M_T <- M_B
22:     else if |M_B| = 0 then:
23:       M_T <- M_A
24:     return M_T
```

Figure 4: Algorithm 2-:The Word2HTN procedure that learns HTNs from Plan Traces

We select the landmark from the last grouping done by HAC on the vocabulary of all the plan traces. Using this landmark, we split the traces that contain the landmark word (action or atom) into the before and after traces. Then we repeat the process on each of the subset of traces, and proceed recursively until a trace set has one or no actions. This base case is discussed in further detail later in this section. Overall, what the algorithm does is divide the overall task into two parts by using the landmark. These are the goals achieved before the landmark, and the goals achieved after the landmark.

The first landmark selected from a trace set may *not* be present in all the traces. We track the subset of traces that does contain the landmark and add it to the set of *processedSubsets* (Line 8 and line 9 of Algorithm 1). When we select the next landmark , we first check if the subset of traces that contain the landmark has already been processed, i.e. is contained in *processedSubsets* (Line8 of Algorithm1). If so, then we ignore the subset. For comparing trace subsets, we use pythons hashing function, which is implemented as a low-level C function, and thus optimized for speed.

Iterating through the landmarks (Line 6) in order, we select a landmark and get the subset of traces that contain the landmark (Line 7). For this subset of traces we call the procedure *ConstructMethods* passing traces and the common landmark l (Line 10). It is the *ConstructMethods* function that divides the traces by the landmark, and recursively calls itself on progressively smaller trace subsets.

In the *ConstructMethods* procedure, we divide the set of traces into those left of the landmark *l* and those right of *l* (Lines 12 and 13). For each of them, we rerun the

HAC on the vector representations of just those words that are in the subset of traces (Lines 14 and 15). From the results of the HAC, we obtain an ordered set of landmarks (*A* and *B*). Then we perform a recursive call on the traces that contain each of the landmarks (Lines 23 and 31). The recursive call continues the decomposition until the base case of a single or no actions in the plan traces (Line 21 for *A* and Line 29 for *B*). If there are no actions in the plan traces, then the procedure MakeMethods returns no methods. If there is only one action, then MakeMethods will return the method representation of the low-level action (Line 21). The method made from the action will have the associated task's goals set as all the effects of the action. The method's preconditions and the task's preconditions are set to the action's preconditions. Finally, the method's subtasks would be the primitive action itself. So, if a planner executed this low-level method, it would result in the action being executed.

After the base case of the recursion completes, it will return to the parent iteration. The parent iteration will then combine the lower level methods as in Line 32 shown in Algorithm 1. The *MergeMethod* function is detailed in Algorithm 2. Each method m_a from the traces to the left of the landmark is combined with each method m_b to the right of the landmark (Lines 3 and 4 of Algorithm 2). The way we combine two methods (m_a, m_b) into a higher-level merged method (m_t), is by looking at common objects between m_a and m_b. We first determine what objects *C* are common across the tasks of the two methods (Line 7). Then we define the goals for the task of m_t as the union of the goals of the tasks of m_a and m_b (Line 10), and are only the goals that have the common objects in *C* (Lines 8 and 9). We also remove from the goals of m_t's task, the atoms that were changed or deleted by the subtask m_b (Lines 11 and 12).

The preconditions of m_t are the union of the preconditions of m_a and m_b (Line 13) minus the goals of m_a's task (Line 14). We do this because the goals of task m_a may satisfy some of the preconditions of m_b. Finally, the subtasks of m_t are the tasks of m_a and m_b in that order (Line 16). The merged method m_t is added to the set M_T (Line 17).

The set of learned methods M_T is added to the *AllMethods,* and returned to the calling function in Line 32 of Algorithm 1. We also add into AllMethods, m_a and m_b (Line 18 and 19). This ensures that the higher-level method has lower-level methods that achieve its subtasks.

At the end of the iteration through the landmarks at the top level in Algorithm 1, we will have a library of HTN methods with a semantically relevant decomposition of tasks. We see this semantic relevance, because the clustering and subsequent division of traces was based on semantic vector representations.

The insight that we used, is that the landmarks can be used to build the skeleton of the HTN hierarchy. Once all the methods are learned, we treat the objects as variables. Variables that had the same object name become the same variable.

4.2 Adding Arithmetic Conditions to Preconditions and Effects:

Thus far, we have dealt with symbolic values in the atoms, such as "*Box1 locatedIn Location2"*. We have not used numeric values, or arithmetic operators in preconditions or effects. In fact, there has been no literature thus far involving learning HTNs with arithmetic operators.

We decided to incorporate support for numeric values and conditions. The format we used to indicate that numeric values or arithmetic operators were in use for a value or

condition is of the form *"MATH~<expression>~"*. The expression can take the form of a comparator operation on numeric values such as *">= 4"*. These would be the used to specify preconditions. It can also be an arithmetic operator that changes the existing value such as *"+3"* or *"-2"*. Such expressions would be used to specify the effects. Lastly the expression could just be a single numeric value. Internally, all numeric values are stored as floating point numbers. This was not necessary for the current experiments, but future work using WORD2HTN will necessitate it. The arithmetic operations currently supported are *" > , < , = , >=, <= , + , -"*. We have not yet added support for multiplication or division, due to time constraints. We think the effort will be straightforward, but would probably require significant testing for various cases.

The only change to the learning algorithm with arithmetic operators is how the preconditions and effects add up. The preconditions update typically involves changing the limits of the inequalities towards constraining them further. For example, adding the conditions *" >=2" , "<= 5",* and *">=3"* would result in the condition *"<=5, >=3"*. If there are two conditions that conflict or cannot be satisfied, then the algorithm would output an error. As for the effects, they are updated by simply combining the operations in the order in which they occur. For example *"+3"* and *"-2"*, would result in *"+1"*.

Chapter 5: Experimental Evaluation

All experiments were run on a Linux Ubuntu 16.04 on an Intel Core machine with i7-4770 CPU running at 3.40GHz with 16 GB of memory. For testing and comparing WORD2HTN we used 3 domains. Logistics, Abstract Graph, and the Malmo interface for Minecraft domain. All domains were deterministic.

5.1 Logistics Domain

We setup a version of the logistics transportation domain [Veloso, 1992]. In our logistics domain setup, there are two planets, 4 cities inside planet1 and 1 city in planet2, and 3 locations within each city in planet1. There is a box in a starting location in planet 1, and the goal is to deliver the box to Planet 2's city1-location1 (P2C1L1). To transport the box, the planner can use trucks, airplanes and rockets. Trucks can move to any location within a city. Airplanes can only fly to locations that are airports, and to cities within a planet. Rockets can move between locations that are spaceports, and can move across planets. Each vehicle can perform the actions of *move, load and unload*. The input plan traces were for transporting the box starting in random locations inside planet 1 to the destination location in planet 2. These were fed into the WORD2HTN algorithm for learning.

We used the Skip-Gram architecture for the word2vector component of the WORD2HTN algorithm. We chose it, as the word embeddings learned were better in terms of reflecting the semantic relationships and similarity of the atoms and actions. The number of dimensions N parameter in the word2vector component was set to half the number of unique atoms and actions in the plan traces. This worked out to be 38 for our

experiments. This is quite a generous rule of thumb. Generous in the sense of dimensional space in which the vector representations can distribute themselves. Increasing the number of dimensions, exponentially increases the time taken to train the vector representations. It should be noted that the entire Wikipedia corpus could be processed effectively with 400 dimensions. So that is our theoretical upper bound for very large domains. We set the window size to 10, so the context of a word would be 5 words on either side of it. The learning rate was set to 0.001, and so the vector representations would only be adjusted by this amount per data point. The previous two parameters were determined from experimental analysis of the similarity of word vectors and comparing them with our expectations. The same learning rate worked for the other two domains with differing number of atoms and actions as well. The last important parameter in word2vector is the number of iterations that the input data was repeated to allow the vector representations to converge to their final values. We set this to 40. Experimentally, we found that around 20 iterations were adequate, and so we set it to 40 to be safe.

The training plan traces were generated from hand coded methods on the logistics domain. We generated 14 different plan traces for training. All the training traces transport the package from a location in one city, to a location in another city. These plan traces were optimal plan traces for transporting the package between each starting and goal location. With this we generated HTN methods, which the planner then used to solve problems in the domain.

With the logistics domain, we compared the depth of the learned HTNs with that of HTN-Maker [Hogg,2008]. HTN-Maker was the previous state-of-the-art HTN learner.

It only learned methods using a right-recursive approach to combining actions and building methods. This mean that it would combine the last two actions to learn a method, and then proceed backwards by combining every action with the previously learned method's task into a new method. This would result in a deep HTN tree structure. In contrast, our approach divides traces by landmarks, and learns methods for each section of the traces. So, we expected the HTN tree to be smaller when solving problems with the learned methods. We compared the HTN tree that resulted from planning for different problems, and each problem had a different optimal plan length. The results are discussed in chapter 6.

5.2 Abstract Graph Domain

The next metric we looked at, was the time cost of planning in domains that were information-dense. We say information-dense in the sense that there are significantly more properties and actions in the domain, than is necessary to achieve the desired goals. It can be interpreted as noise, or relevant for different goals.

The comparison that we did was with Fast Downward, the classical planner that uses Planning Domain Description Language (PDDL). It is the current state-of-the-art for classical planners, and has different heuristics to optimize the planning process.

To compare the effects of information density on planning time for WORD2HTN, and compare it to Fast Downward, we developed the Abstract Graph domain. This domain was developed to be extensible and configurable with a simple set of parameters. As the name implies, it is a graph based domain. In it, the goal is for a robot to navigate

between nodes. Every node in the graph can have additional properties that may or may not be relevant to an action in the domain. For example, figure 5 shows the subgraph for nodes that have the properties of x and y co-ordinates. On such nodes, we can perform an action such as "*move+X+Y+node1+node2*", and the robot will move from node1 to node2 and update its *X* and *Y* values to that of node2. There could be additional properties in the domain, but the example action specifically updates only the X and Y property.

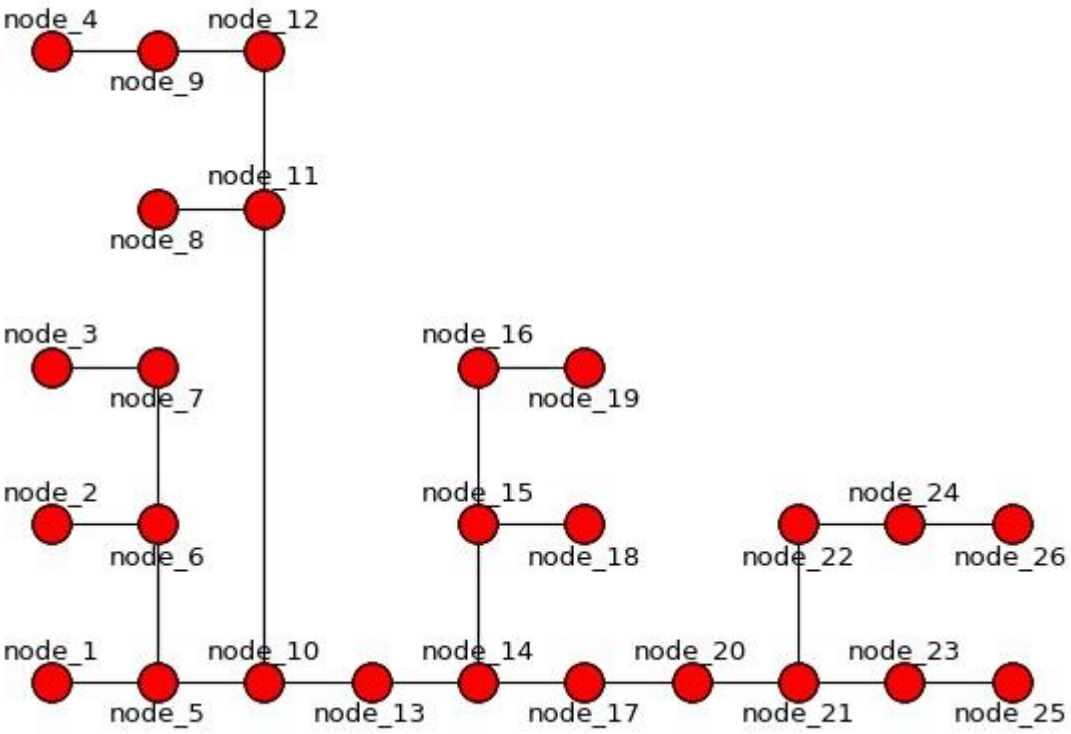

Figure 5: Subgraph with (x,y) coordinates as the only properties

We trained WORD2HTN on plan traces from Figure 5, whose initial location was a node from a set of nodes1…12 to a node from the set of nodes13…26. The plan traces also included the coordinate properties (X and Y) of the nodes as the robot visits them.

The plan traces do not have any additional properties. However, in the problem space there are additional properties that the nodes can have. The number of these properties was modified by a script. These additional properties affect the state space of the domain (i.e., possible actions and atoms). There is an operator in the domain to change every combination of properties in the robot as it moves through the nodes. This means that if there were only 2 properties, then there would be 4 move operators. With 3 properties, there would be 8 operators and so forth. Figure 6 shows the graph that we randomly generated for our testing. For the experiments, the graph had 100 random nodes with a 12.5% chance that any two nodes are connected. We varied the number of properties in the nodes to see how the Fast Downward planner with three different heuristics would compare with WORD2HTN. Internally, WORD2HTN uses a SHOP planner [Nau et al., 1999] with the learned methods to solve tasks. Our comparison results are in the following chapter on results.

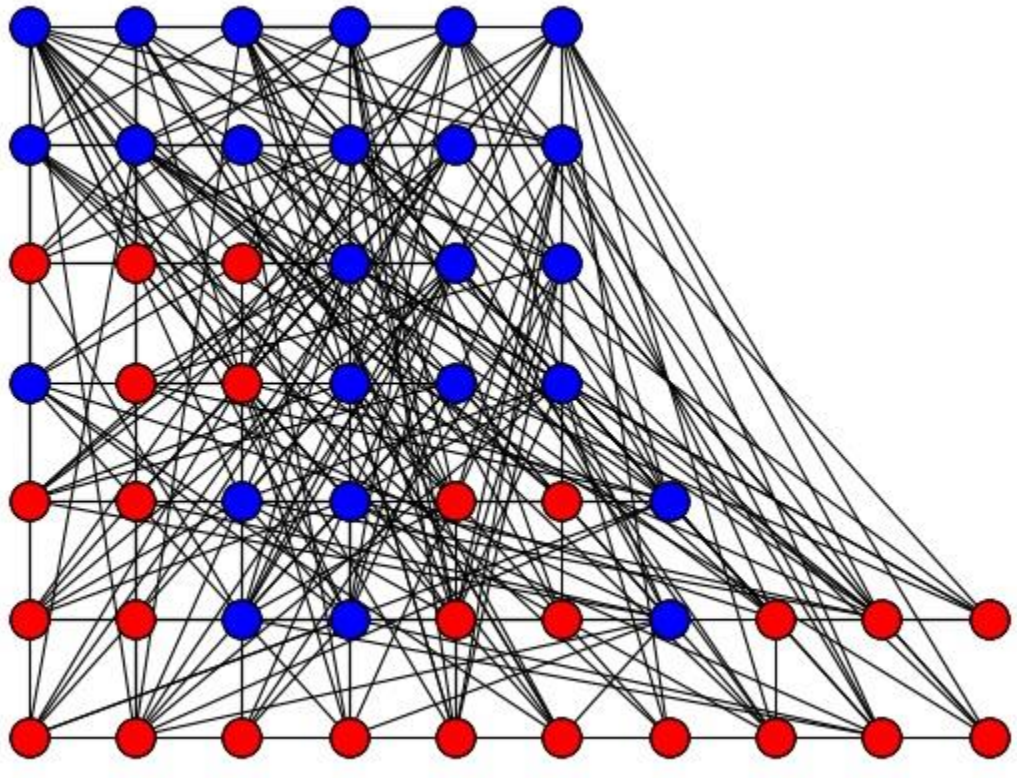

Figure 6: Complete Graph for Abstract Domain Experiment with random nodes and connections

5.3 Minecraft experiments with arithmetic preconditions and effects:

The final domain that we worked with is the game Minecraft, using the Malmo interface for the game [Johnson et al., 2016]. Minecraft is a game where the player can control an agent in the game to collect resources, and build objects. For example, the player can build a furnace by collecting and combining 8 stone as show in Figure 7.

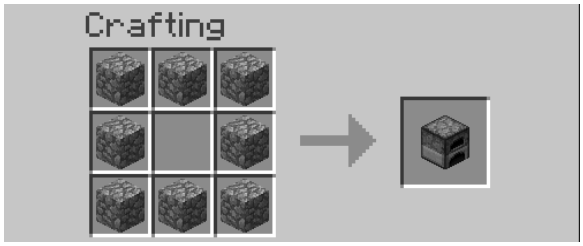

Figure 7: Minecraft crafting example - combining 8 stone into a furnace

There are many such *recipes* in the game. Successive recipes or items have dependencies to achieve or build the object. For example, to harvest the stone needed to make a furnace, the player must first have a wooden pickaxe to harvest the stone. For a list of the basic dependencies, see figure 8

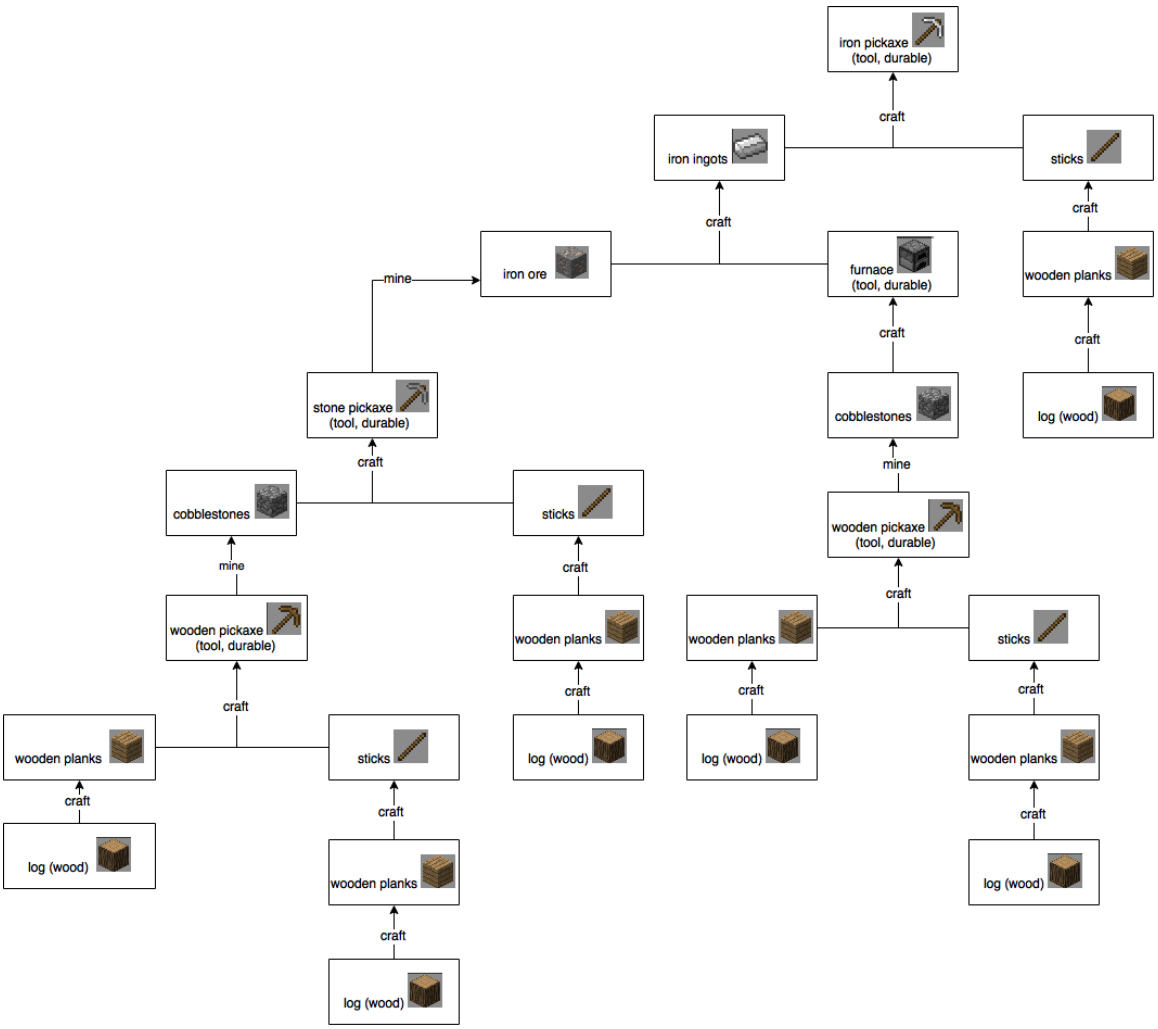

Figure 8: Minecraft basic recipes and dependencies [Nyugen S., 2017]

Using the Malmo interface, we were able to generate a closed world with randomly placed resources. The main objective of this phase of the work was to have an automated *assistant* agent as shown in figure 9 to help the main Minecraft agent complete tasks fasters (cooperative play).

Figure 9: Minecraft main player and assistant agent

The agent learns by observing the main player complete tasks. The main player's actions are translated into plan traces. Those plan traces are the input into WORD2HTN algorithm. An example of an action in the plan trace would be *"... , hasStone(8), CraftFurnace, hasFurnace(1), …"* for the action *CraftFurnace*.

Once WORD2HTN learned the library of methods, it could then be queried for a plan to achieve a goal. For example, one could ask the Planner to solve for the goal *"hasIronPickaxe(1)"* and pass in the current state of the agent. WORD2HTN would then generate the plan to achieve the goal. For WORD2HTN to work with Minecraft's dependencies and recipes, we needed arithmetic conditions and effects as described in section 4.2.

We tested the time taken for an agent to complete a task on its own, versus two agents (cooperative play). The task was to make two iron pickaxes, and the experiment was run 10 times. The results are described and discussed in the following section on Results.

Chapter6: Results

6.1 Logistics Domain Results:

The data we observed comparing the maximum tree depth of the HTN learned by WORD2HTN with that of a right-recursive method learning algorithm like HTN-Maker is captured in Figure 10. The optimal tree depth for a plan trace of *n* actions should be $log_2 n-1$ for an even binary division of actions at each level. In the graph, there are no plan traces shorter than 3 actions, because in the Logistics domain that is the minimum number of actions needed to transport a package between one location and the nearest adjacent location. For the logistic domain that we used, WORD2HTN was able to learn the optimal division of actions, i.e. it selected landmarks that evenly divided the plan traces. We don't always expect the result to be an optimal division, only that it will be divided by semantic relevance. This result occurred because the problem was very symmetrical and the landmarks could divide the plan traces evenly. The data validates our hypothesis that HTNs learned are balanced in comparison to a right-recursive method structure such as those learned by HTN-Maker.

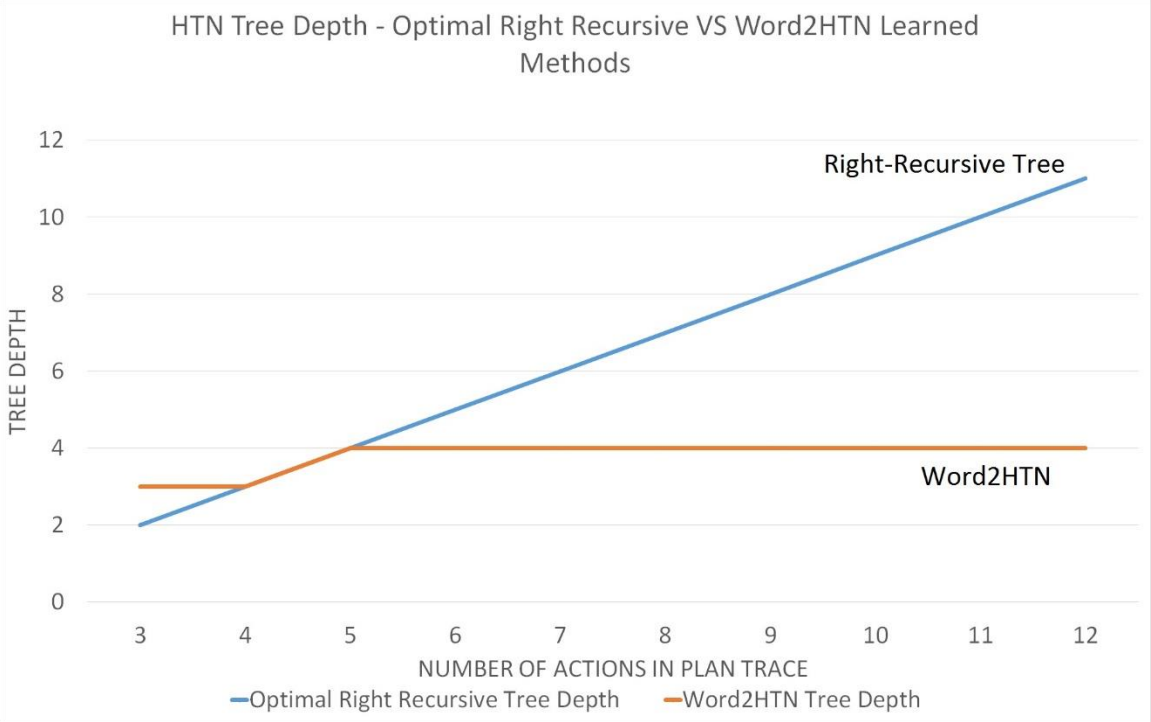

Figure 10 : Comparison of the Tree Length of HTNs learned by WORD2HTN with those of a right-recursive learning algorithm

In addition, the learned methods could solve more problems than those with which they were trained. We had trained on 12 problems, and could solve all other problems in the domain. This result is only because all other problems in the domain involve the same structure, or substructure that the training plan traces encompassed, and that the domain had common substructures, or very symmetrical. Here we say symmetrical because the problem of transportation involves the same subset of actions for transportation within a city, between cities, and across planets. We clarify this to be accurate about the limits of WORD2HTN.

6.2 Abstract Graph domain Results

Figure 11 shows the results for our runtime comparisons, as a function of the number of properties in the domain. Each data point is the average runtime over 12 runs for 4 test planning problems each of which were run 3 times.

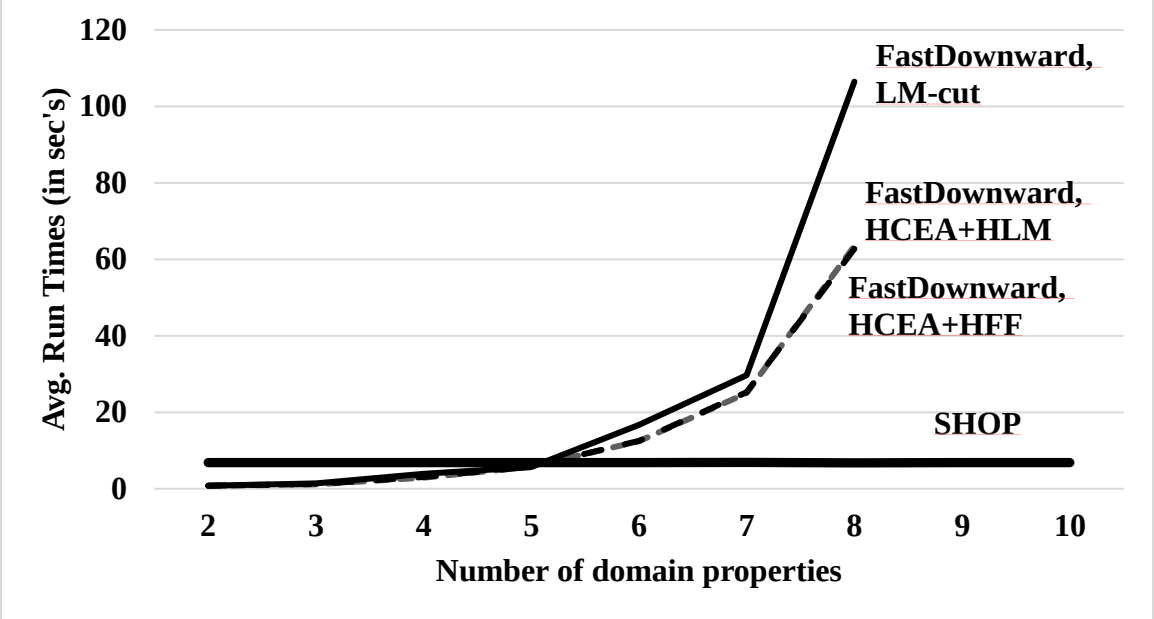

Figure 11 : Comparison of the time taken by Fast Downward using 3 different heuristics versus the SHOP planner with learned HTN in the Abstract Graph Domain

The SHOP planner using the learned HTNs took the same amount of time regardless of the number of additional properties in the domain as the HTN's learned would only consider the X and Y coordinates of the nodes (relevant properties). As for Fast Downward, initially it was faster than the SHOP planner, but as the number of properties (and thus actions and problem space) increases, the time taken grows exponentially. Fast Downward could not solve the test planning problems that had 9 or more domain properties; the potential states and action space was so large, that it exhausted the

memory of the testing computer during the pre-processing phase of the problem. Thus, Figure 4 does not show any data points for those cases. In these experiments, we observed that Fast Downward usually generated shorter plans than SHOP. In comparison, SHOP did not necessarily generate the shortest plans using the WORD2HTN-learned knowledge. Instead, WORD2HTN was able to learn a semantically relevant decomposition of the plan traces and convert them into HTNs. With that knowledge, SHOP could solve the planning problems with an exponential reduction in run times as compared to Fast Downward.

6.3 Minecraft Results

We ran the experiment to compare the time taken for single agent to make 2 iron pickaxes, and compared it to the time taken for two agents to perform the same task. The results are presented in Figure 12 [Nyugen et al. 2017]. On average, the time taken for two agents to make the iron pickaxes was 147.4 seconds and the median runtime was 147.5 seconds [Nyugen et al. 2017]. For a single agent to make both pickaxes, the average time taken was 174.2 seconds, and the median time was 174 seconds. Thus, two agents completed the task 16% faster on average. This shows the promise and benefit (speed improvement) of using the plans generated from the WORD2HTN methods for controlling an assistant agent

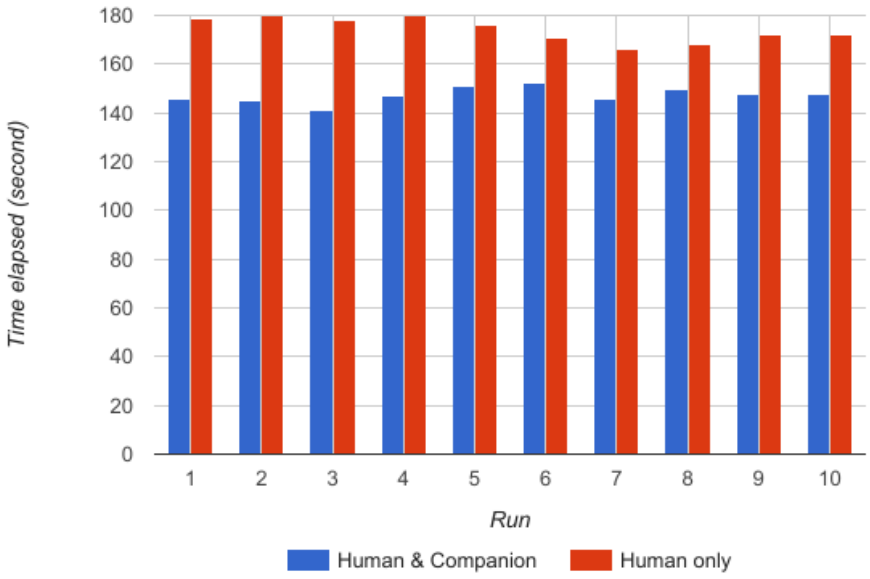

Figure 12: Comparison of time taken to build two pickaxes with and without an assistant agent

Chapter 7: Related Work

People learn sophisticated skills by building upon basic or simpler actions and skills. These are combined in a particular order to achieve higher or more complex tasks [Choi and Langley, 2005]. A hierarchical structure lends itself very nicely to represent the dependencies and build-up of more complex skills or methods to achieve a task. Such hierarchical representations are used in several frameworks for reasoning and planning such as frames [Minsky 1986], reasoning by abstraction [Knoblock, 2012], and hierarchical task network (HTN) planning [Currie and Tate, 1991]. Most hierarchical planning and reasoning formalisms have concepts in common such as objects, goals and tasks. These concepts are used to represent the knowledge and the relationships in the hierarchical structure.

Hierarchical Task Networks is the formalism that we use. Learning HTNs has two major components:

(1) Learning the hierarchical decompositions that relate the tasks to subtasks

(2) Learning the conditions in which, the task can be achieved using a specific decomposition. This is a method.

There is prior work that focuses on learning the applicability conditions (preconditions) for the decomposition, but assume that the hierarchical decomposition information/structure is part of the input into the algorithm [Ilghami et al., 2005; Hogg et al., 2014]. This means that the subtasks of a task are known. Most work on HTN learning learn both, the decomposition hierarchy and the conditions for decomposition. These

works infer hierarchy from the set of input plans, and from the action model of the domain [Zhuo et al., 2014; Hogg et al., 2014; Choi and Langley, 2005; Reddy and Tadepalli, 1997; Ruby and Kibler, 1991].

However, all the work done on learning task structures assume that some task semantic information is given as additional input. Here, we mean task semantics as defined for annotated tasks in the preliminaries section. The task semantics are the goals, preconditions, and header(name) of the task. The goals and preconditions of such annotated tasks are the minimum requirements to satisfy or complete the task. Often methods that are learned to satisfy a task, may have more than this minimum set.

Among the prior work, X-Learn uses inductive generalization to learn task decomposition. It relates goals, subgoals, and conditions for applying these decomposition rules[Reddy and Tadepalli, 1997]. X-Learn takes task information as part of the input along with the plan traces for the learning problem. This task information is not needed in WORD2HTN.

ICARUS is another learning algorithm that receives as input, horn clauses (called skills) that define the semantics of the goals (subgoals, and preconditions). These skills are used in a teleo-reactive process to learn the hierarchies [Choi and Langley, 2005; Nejati et al., 2006].

HTN-Maker receives as input, tasks whose semantics are also specified. These are the previously defined annotated tasks, (τ,p,g).[Hogg et al., 2008; 2014]. HTN-Maker uses the annotated tasks to detect sub-traces such that the task's preconditions are satisfied when the sub-trace starts and the task's goals are achieved when the sub-trace

ends. Subtask to Task relations for the hierarchy are detected when a sub-trace identified for one task is itself part of a larger trace that satisfies another larger task.

HTN-Learner also uses annotated tasks as input [Zhuo et al., 2014]. It constructs constraints such as "*task's precondition is satisfied before action a in trace 25*". These constraints are then fed into a MAXSAT solver to generate solutions for the constraints which are used to generate the task decompositions and the preconditions. Unlike all these works WORD2HTN learns the annotated tasks (semantic/defining information of the tasks), in addition to the hierarchical structure, and the applicability conditions.

With regards to arithmetic conditions that are learned for HTNs, this is the first work to the best of our knowledge to learn such conditions. Classical planners that employ PDDL (Problem Domain Description Language) version 2.1 can solve problems that have continuous numeric values, and employ arithmetic operators and comparators [Coles et al. 2012]. Such work does not learn HTNs for learning task relationships and planning. Rather, such classical planners use the action models for heuristically expanding the search space to achieve goals. The downside of this approach is that no knowledge is saved and reused in similar domains. Rather each new problem space is treated anew, and this is typically slower than a HTN planner with a library of HTN methods.

Chapter 8: Conclusions and Future Work

81. Summary of Results and Conclusions

WORD2HTN is an HTN learning algorithm that can find landmarks from plan traces and use them to divide tasks and make HTNs. Internally, WORD2HTN uses Word2Vector to learn semantic vector embeddings. The embeddings are used to infer similarities between atoms and actions, and identify landmarks based on the similarities. Unlike other HTN learning algorithms, WORD2HTN does not require task semantics as part of the input. Only input plan traces, and the domain operators are needed.

WORD2HTN was tested on the Logistics domain, Abstract Graph domain, and a restricted environment of the Minecraft game. We observed that the division of tasks in the HTNs learned was semantically driven. The depth of the HTN tree was less than the HTN made from a right recursive combination of actions. The latter is what the current state of the art HTN learner does. We also observed that the learned HTNs could solve more problems than those given in the training data.

In the Abstract Graph domain, the SHOP planner using our learned HTNs solved problems faster than Fast Downward planner in problems that were information dense (had extra information and actions).

Finally, we observed that the HTNs learned could be used to control a cooperative agent in Minecraft and reduce the time taken to complete a task.

8.2 Future work

In this work, all the algorithms and experiments were done in deterministic domains. In non-deterministic versions of the experimental domains, each action has a probabilistic distribution of outcomes. For example, a *loadTruck* action may result in the package being broken, or not loaded onto the truck. It is necessary to test WORD2HTN algorithm in the non-deterministic domain for it to be useful, and so that will be the next phase of this work.

In non-deterministic domains, we believe that similar atoms and actions will still be grouped closer together, and the landmarks will still be found as in deterministic domains. The difference is that there are more atoms that will result from actions, and so the atoms and action's vector representations will not be as close together as in the deterministic domains. We think that the landmarks will still be discovered, as the relative similarity will be preserved.

We would also like to improve the support of arithmetic conditions and effects in WORD2HTN. The next step is to add multiplication and division to describe more domains. Adding additional arithmetic support will be an ongoing work, with additional support added as needed.

After adding non-deterministic support, it is important to tag or qualify actions, atoms, and methods with metrics. This could be metrics like a reward value or cost. There could be more than one metric, such as time, resources used, and other such measurements. These metrics would be associated with the methods learned. To learn these values for hierarchical methods, we would need to define the functions that calculate each of the metrics. We can then use these metrics to define additional

requirements or bounds when we specify tasks or goal states. This will help the planning by constraining, and directing the planner's search through the HTN methods.

Bibliography

[Choi et al., 2005] Choi, D., & Langley, P. (2005, August). Learning teleoreactive logic programs from problem solving. In International Conference on Inductive Logic Programming (pp. 51-68). Springer Berlin Heidelberg.

[Coles et al. 2012] Coles, A. J., Coles, A. I., Fox, M., & Long, D. (2012). COLIN: Planning with continuous linear numeric change. Journal of Artificial Intelligence Research, 44, 1-96.

[Currie and Tate, 1991] Currie, K., & Tate, A. (1991). O-Plan: the open planning architecture. Artificial Intelligence, 52(1), 49-86.

[Erol et al., 1994] Erol, K., Hendler, J., & Nau, D. S. (1994, October). HTN planning: Complexity and expressivity. In AAAI (Vol. 94, pp. 1123-1128).

[Fox and Long, 2003] Fox, M., & Long, D. (2003). PDDL2. 1: An extension to PDDL for expressing temporal planning domains. J. Artif. Intell. Res.(JAIR), 20, 61-124.

[Gopalakrishnan et al., 2016] Gopalakrishnan, S., Munoz-Avila, H., Kuter, U., (2016) WORD2HTN: Learning Task Hierarchies Using Statistical Semantics and Goal Reasoning. In Proceedings of the 25th international joint conference on Artificial intelligence: Workshop on Goal reasoning.

[Helmert et al., 2006] Helmert, M. (2006). The Fast Downward Planning System. J. Artif. Intell. Res.(JAIR), 26, 191-246.

[Helmert et al., 2011] Helmert, M., Röger, G., Seipp, J., Karpas, E., Hoffmann, J., Keyder, E., ... & Westphal, M. (2011). Fast downward stone soup. Seventh International Planning Competition, 38-45.

[Hogg et al., 2008] Hogg, C., Munoz-Avila, H., & Kuter, U. (2008, July). HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required. In AAAI (pp. 950-956).

[Hogg et al., 2014] Hogg, C., Muñoz-Avila, H., & Kuter, U. (2014). Learning hierarchical task models from input traces. Computational Intelligence.

[Ilghami et al., 2005] Ilghami, O., Munoz-Avila, H., Nau, D. S., & Aha, D. W. (2005, August). Learning approximate preconditions for methods in hierarchical plans. In Proceedings of the 22nd international conference on Machine learning (pp. 337-344). ACM.

[Knoblock, 2012] Knoblock, C. (2012). Generating abstraction hierarchies: An automated approach to reducing search in planning (Vol. 214). Springer Science & Business Media.

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

[Johnson et al., 2016] Johnson, M., Hofmann, K., Hutton, T., & Bignell, D. (2016). The malmo platform for artificial intelligence experimentation. In the proceedings of the 25th International joint conference on artificial intelligence (IJCAI) (p. 4246).

[Mikolov et al., 2013] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

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

[Minsky, 1986] Minsky, M. (1986). The Society of Mind, Simon 8.

[Nejati et al., 2006] Nejati, N., Langley, P., & Konik, T. (2006, June). Learning hierarchical task networks by observation. In Proceedings of the 23rd international conference on Machine learning (pp. 665-672). ACM.

[Nau et al., 1999]]Nau, D., Cao, Y., Lotem, A., & Munoz-Avila, H. (1999, July). SHOP: Simple hierarchical ordered planner. In Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2 (pp. 968-973). Morgan Kaufmann Publishers Inc.

[Nau et al., 2005] Nau, D., Au, T. C., Ilghami, O., Kuter, U., Wu, D., Yaman, F., ... & Murdock, J. W. (2005). Applications of SHOP and SHOP2. IEEE Intelligent Systems, 20(2), 34-41.

[Nyugen S., 2017] Nyugen S. (2017) Minecraft basic recipes and dependencies. JPEG. Lehigh University.

[Nyugen at al. 2017] Nyugen, S., Reifsnyder, N., Munoz-Avila, H., Gopalakrishnan, S. (2017). Automated Learning of Hierarchical Task Networks for Controlling Minecraft Agents. <***under review***>.

[Reddy and Tadepalli, 1997] Reddy, C., & Tadepalli, P. (1997, July). Learning goal-decomposition rules using exercises. In AAAI/IAAI (p. 843).

[Rong, 2014] Rong, X. (2014). word2vec parameter learning explained. arXiv preprint arXiv:1411.2738.

[Ruby and Kibler, 1991] Ruby, D., & Kibler, D. F. (1991, July). SteppingStone: An Empirical and Analytical Evaluation. In AAAI (pp. 527-532).

[Veloso, 1992] Veloso, M. M. (1992). Learning by analogical reasoning in general problem solving (No. CMU-CS-92-174). CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

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

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

Curriculum Vitae:

EDUCATION:

M.Sc. Computer Science from Lehigh University , Bethlehem, PA

2015-2017

B.Sc. E.C.E, and B.A. in Computer Science from Lafayette College, Easton, PA

2005-2010

PUBLICATIONS:

Gopalakrishnan,S., Muñoz-Avila,H., Kuter,U., Word2HTN:Learning Task Hierarchies Using Statistical Semantics and Goal Reasoning. The IJCAI-2016 Workshop on Goal Reasoning. AAAI Press. 2016. New York,NY,USA.

Yu,Y., & Gopalakrishnan,S. Elastance Control of a Mock Circulatory System for Ventricular Assist Device Test. In proceedings of American Control Conference. 2009. St Louis, MO. USA.1009-1014.

Yu,Y., & Gopalakrishnan,S. Evaluation of a minimally invasive cardiac function estimator for patients with rotary VAD support. In proceedings of IEEE 33rd North East Bio Engineering Conference. 2007. Long Island,NY,USA.161-162.

ACADEMIC HONORS:

Dean's Doctoral Assistantship, Lehigh University

2016-2017

J.J. Ebers memorial award for outstanding achievement in E.C.E. (Lafayette College)

05/2009

2nd Place at the David and Lorraine Symposium for Undergraduate Research (at Lehigh University)

04/2009

Honor Societies: Tau Beta Pi (Engineering), Eta Kappa Nu (Electrical Engineering) Upsilon Pi Epsilon (Computing and Information Disciplines), Phi Beta Kappa (Liberal Arts and Sciences)