Learning Decomposition Methods with Numeric Landmarks and Numeric Preconditions

Morgan Fine-Morris[1], Michael W. Floyd[2], Bryan Auslander[2], Greg Pennisi[2], Kalyan Gupta[2], Mark Roberts[3], Jeff Heflin[1], Héctor Muñoz-Avila[1]

[1]Department of Computer Science, Lehigh University 113 Research Dr., Bethlehem, PA 18015, USA
[2]Knexus Research Corporation, 174 Waterfront Street, Suite 310, National Harbor, MD 20745 USA
[3]Navy Center for Applied Research in AI, Naval Research Laboratory, Washington, DC, USA
[1]mof217@lehigh.edu, {heflin, munoz}@cse.lehigh.edu, [2]{first.last}@knexusresearch.com, [3]mark.roberts@nrl.navy.mil

Abstract

We describe an HTN method-learning system, which we call T2N, that learns hierarchical structure from plan traces in domains with numeric effects, where some subgoals are numeric. We investigate how different methods of preprocessing training data can impact the effectiveness of the learned methods. We test the learned methods by solving a set of 30 test problems in a simple numeric crafting domain based on the videogame Minecraft. Our results indicate that we can learn functional methods for domains with these characteristics and suggest that different preprocessing techniques lead to method sets with different strengths and weaknesses, with no preprocessing technique superior across all domain tasks.

1 Introduction

Hierarchical Task Network (HTN) planning is a planning paradigm where the planner takes advantage of user-provided domain information (called decomposition methods) to guide the search process. It decomposes complex tasks into progressively simpler ones, until it reaches the granularity of primitive tasks, accomplishable directly by the domain actions. It can provide significant speed up over classical planning techniques, at the cost of extra up-front knowledge engineering required to define the decomposition methods (Ghallab, Nau, and Traverso 2004; Nau et al. 2001) A major problem in applying HTN planning to new domains is the expense and difficulty of authoring correct and useful domain information. A previous system (Fine-Morris et al. 2020) for learning decomposition methods for domains with numeric preconditions was unable to deal with domains where all effects (and therefore goals) were numeric. In this work, we describe a system (called T2N, for Trace2NumericHTN) that, while based on similar design principles, can learn HTNs where changes in numeric variables dictate the structure of the network.

To create structure for our learned HTNs, we identify conditions, called bridge atoms (Gopalakrishnan, Munoz-Avila, and Kuter 2018; Fine-Morris et al. 2020), that mark switches in context within the traces and which we describe as "domain" landmarks. We use these conditions as subtasks into which our learned decomposition methods decompose the high-level tasks of a a domain. Hereafter we use bridge atom and landmark atom interchangeably.

We learn several sets of decomposition methods, each with a different set of domain landmarks as subtasks, then evaluate their performance by solving 30 planning problems. Our results indicate that the set of domain landmarks can impact planning efficiency and goal coverage, and that variations in the way we preprocess the traces before domain landmark learning can impact domain landmark selection, resulting in downstream effects on the learned methods. While more data is necessary to draw a final conclusion about the best way to select domain landmarks, this work shows that comparing method performance on a set of test problems can help us analyze the utility of different sets of domain landmarks.

2 Background

We will provide background information on HTN planning, planning and domain landmarks, extracting domain landmarks, and learning numeric preconditions using goal regression over numeric actions.

2.1 HTN Planning

An HTN planning problem, P, is a triple (D, s_0, T), where D is a domain description, s_0 is an initial state, and T is a list of tasks to accomplish. The domain description D is a tuple (O, M), where O is the set of domain operators and M are the decomposition methods. An operator $o \in O$ is a tuple (h, p, e), where h is the head (i.e., the task name and arguments), p are the preconditions, and e are the effects. A method $m \in M$ is a tuple (h, p, sub), where h and p are the same as for an operator, and sub is an ordered sequence of subtasks. Primitive tasks are those that are accomplishable using operators in O (ground versions of the head of an operator), while complex tasks must be further decomposed by the methods in M until they are reduced to a sequence of primitive tasks that can be performed to accomplish the complex task. What constitutes 'accomplishing' a task is not formally defined. In this work, we learn tasks that are concerned with achieving a goal and use them with an HTN planner. Therefore, we straddle two different hierarchical planning paradigms, HTN planning and Hierarchical Goal Network (HGN) planning, which is very similar to HTN planning, except that instead of decomposing tasks into subtasks, goals are decomposed into subgoals (Shivashankar et al. 2012).

2.2 Planning and Domain Landmarks

Planning landmarks are conditions which always occur in the solution of a given problem and have been addressed in many works (Hoffmann, Porteous, and Sebastia 2004; Porteous, Sebastia, and Hoffmann 2014). We are interested in finding a type of landmark that we call *domain landmarks*, common conditions for two or more problems in a domain as opposed to *planning landmarks*, which are conditions common to one problem in a domain. Hereafter any reference to "landmarks" instead of "planning landmarks" refer to domain landmarks.

Given the set of all problems in a domain, Φ, we define a ϕ-domain landmark as a common planning landmark for every problem $P \in \phi$ where $\phi \subseteq \Phi$. Confirming a planning landmark for a problem is PSPACE-complete problem (Hoffmann, Porteous, and Sebastia 2004) and confirming a domain landmark will increase the running time by the factor $|\phi|$. Therefore, we offer a 'pragmatic' definition of ϕ-domain landmarks: any condition that occurs in at least one solution to every problem in ϕ. In our work, ϕ consists of problems that have a similar initial state s_0 and/or list of tasks T, such that the problems will have common conditions for at least one solution for each problem in ϕ.

As in (Fine-Morris et al. 2020), we use these domain landmarks as subtasks for the tasks demonstrated in the training traces. We also use these domain landmarks to partition the traces into subtraces from which we learn methods for accomplishing each domain landmark. Unlike (Fine-Morris et al. 2020), in this work we learn a two level hierarchy of methods. The landmark methods decompose single top level tasks of the domain into subtasks based on the domain landmarks. These landmark tasks are decomposed by *subplan methods* into sequences of primitive tasks that achieve the landmark. For a generic example hierarchy learned from a trace containing three landmarks, see Figure 1. A trace may not contain all selected domain landmarks, but as in (Fine-Morris et al. 2020) we partition traces at most once per landmark, therefore the maximum number of partitions per trace is $|LM| + 1$ for the set of selected domain landmarks, LM.

Note that landmark methods derive their name from the use of landmarks to decompose the problem, **not** because they show the planner how to accomplish/achieve any individual landmark (except inasmuch as the achievement of later landmarks may be dependent on the achievement of earlier landmarks). They select which landmarks need to be achieved as subgoals of the final task. The role of determining *how* to achieve a landmark falls to the non-landmark methods which, unlike landmark methods, can have primitive subtasks.

2.3 Domain Landmark Extraction

We approximate the domain landmarks using a technique similar to the ones used in (Gopalakrishnan, Munoz-Avila, and Kuter 2018; Fine-Morris et al. 2020). This involves using the natural language processing algorithm Word2Vec (Mikolov et al. 2013) to learn a word embedding for each word (a condition atom or action) in the corpus (a set of traces). Word embeddings are vectors, each associated with a word, whose direction describes the context in which the word appears in the corpus. Therefore, two words with similar direction vectors can be assumed to co-occur frequently in the corpus. We use these vectors to determine how to split the words in the corpus into separate context regions. We use Hierarchical Agglomerative Clustering to form clusters of words, using cosine distance (a measure of the difference of the angle between two vectors) as the metric so that words which share contexts are clustered together. This creates separate context regions in the domain, so that we can probe the boundaries between the regions for condition atoms that signal a shift between the contexts. These condition atoms are our domain landmarks.

2.4 Learning Numeric Preconditions with Function Composition

In addition to learning the subtasks of methods we also must learn preconditions, which may contain numeric functions. We use a modified form of goal regression presented in Fine-Morris et al. (2020). Traditional goal regression involves inverting the effects of an action, so that the effects of the action are removed from a state, and the action preconditions are added to it. Previous work (Fine-Morris et al. 2020) describes how to augment traditional goal regression to regress actions with numeric fluents via function composition. For example, when learning a method that decomposes into a sequence of actions $a1, a2$ where the preconditions and effects of $a1$ and $a2$ are:

$$\begin{aligned} p^{a1} &= [f(varX, \dots) > 1,\ \dots] \\ e^{a1} &= [varX \leftarrow f(varX, \dots),\ \dots] \\ p^{a2} &= [g(varX, \dots) > 2,\ \dots] \\ e^{a2} &= [varX \leftarrow g(varX,\ \dots),\ \dots] \end{aligned}$$

the methods preconditions would include,

$$f(varX, \dots) > 1,\ g(f(varX, \dots), \dots) > 2.$$

The inequalities from both actions occur in the new set of preconditions, but in addition both $g(\dots)$ and $f(\dots)$ are applied to $varX$ before the inequalities from $a2$ is checked, to ensure that whatever value $varX$ is, it will be > 2 after being updated according to both $f(\dots)$ and $g(\dots)$.

3 Example Domain

The domain used in this paper is a custom domain based on the crafting system of the game Minecraft. Base resources (wood, stone, coal, iron) can be gathered from the world, and used during crafting to make new items. The amount of a resource is described by terms in the form `value(item, amount)`, where `item` is a label for a counter of a particular type of item, and `amount` is a numeric value. When a gather action is used to collect a resource from the world, the world's resource counter is decreased by the specified amount, while the resource counter of the agent who performed the gather action increases by that amount.

Many crafting recipes require an item called a crafting table. Some resources – such as wood – can be gathered by hand or using a tool. Others – such as stone, coal, and iron – can only be gathered using a specific type of tool, e.g., a pickaxe. The grade of the pickaxe dictates what resources it can collect and depends on the material it is made from. Pickaxes made from stronger materials can be used to mine harder resources. From weakest to strongest, the materials are: wood, (cobble)stone, iron. Mineable resources from softest to hardest are: stone, coal, iron. Wooden pickaxes can mine stone and coal. Stone and iron pickaxes can mine iron and anything softer. In typical Minecraft, pickaxes and axes have durability, and can only be used a certain number of times before they break. We elided this aspect of tool-use.

When an agent crafts an item, the value of the counters of the consumable ingredients are decreased according to the amounts called for by the crafting recipe. The agent's counter for the crafted item increases by the batch amount, which is usually 1 but can be more as in the case of the rail item, which can only be made in batches of 16.

Some items, called stations, are not consumable but are monopolized while they are required to craft a recipe. We included the crafting table, where argents craft complex items, and the furnace, which agents fuel with coal to smelt iron into iron ingots.

Some items are more expensive to craft from scratch than others, because they require more steps if the agent starts with an empty inventory. For example, to craft a wooden axe the agent only needs to gather wood and make a crafting table, sticks, and planks, while crafting a stone axe requires all the steps to craft a wooden pickaxe, which the agent must use to gather stone to form the blade of the stone pickaxe. To craft an item that has iron ingots as an ingredient (as in the case of our cart, rail, iron pickaxe, and iron axe goal items), the agent must have a furnace to smelt iron, which can only be gathered using a stone pickaxe. In terms of relative expense, wood items < stone items < iron items.

4 Approach

T2N has two phases: learning domain landmarks and learning methods. We next detail these phases and their steps.

4.1 Phase 1: Learn Domain Landmarks

Phase one has three steps. For a set of traces $\mathcal{T}$ (see Section 5.1 for trace generation): Step (1.1) formats $\mathcal{T}$ into three variations and discretizes numeric fluents, Step (1.2) learns word embeddings for $\mathcal{T}$ using Word2Vec, and Step (1.3) selects bridge atoms that partition $t \in \mathcal{T}$ into subtraces. Previously, Fine-Morris et al. (2020) used a similar technique with some differences in Steps 1.1 and 1.3. To summarize, this work better formalizes how to discretize numeric fluents, investigates the impact of formatting $\mathcal{T}$, and examines selecting domain landmarks for *numeric* subgoals.

Step 1.1: Preprocess Traces. To prepare traces for landmark learning, we first must format them in one of three styles, full (FL), precondition-effect (PE), and randomly-augmented (RA). Traces formatted in the FL style have full states (i.e., they contain static conditions carried over from the initial state). The states of traces formatted in the PE style include only the effects of the previous action and the preconditions of the next. Randomly-augmented

FL	PE	RA
value(a_w, 0)	value(a_w, 0)	value(a_w, 0)
value(w, 4000)	value(w, 4000)	value(w, 4000)
value(a_c, 0)		value(c, 350)
value(c, 350)		
...	...	...
gather(w, 5)	**gather(w, 5)**	**gather(w, 5)**
value(a_w, 5)	value(a_w, 5)	value(a_w, 5)
value(w, 3995)	value(w, 3995)	value(w, 3995)
value(a_c, 0)		value(a_s, 0)
value(c, 350)		
...	...	...
craft(plank)	**craft(plank)**	**craft(plank)**

Table 1: Trace formatting variations used for learning word embeddings (with the original numerics, i.e., 'unskolemized'). Actions are in bold. To save space, variable names have been shortened: 'a' replaces agent, 'w' replaces wood, 'c' replaces (cobble)stone, 's' replaces stick; therefore, variable a_c is a shortening of agent_stone. These shortening are not used in the real traces.

traces are partway between the two, with states the contain all the atoms of the PE traces, plus a small set of randomly selected atoms that would be present in the FL version of the state but not the PE version. See Table 1 for examples of all three, illustrating these differences. In the FL example, the atom `value(agent_stone, 0)` is included in states before and after the action `gather(wood, 5)` despite its extraneousness. In the PE example, atoms which are neither preconditions nor effects of the surrounding actions are excluded, `value(agent_stone, 0)` is excluded from the state because it is neither a precondition nor effect of the actions `gather(wood, 5)` and `craft(plank)` which surround the state. In the randomly-augmented (RA) trace, atom `value(stone, 350)` which describes the amount of free (i.e., not held by an agent) stone in the world is included in the first state randomly, and `value(agent_stick, 0)` is included randomly in the second state, although they are not pertinent to either of the actions adjacent to them.

In FL styles, the vocab size is larger and states are much larger because all true conditions are included. This has repercussions when learning word embeddings, as the longer traces make Word2Vec more expensive and creates more connections between words, as static conditions occur with much greater frequency, and any conditions achieved early in the text are particularly effected. PE style traces limit relationships between words to try to ensure strong relationships form only between actions and conditions that are related by function. RA traces try to take advantage of the strengths of both.

In a process similar to skolemization (Bundy and Wallen 1984), we replace the numeric values with names for the value ranges. This ensures that landmarks from variables that take on many values correspond to any value within a range, instead of to a single value. To do this we collect the values for all variables across all traces and then make a histogram for each variable. Then, for each instance of that variable in an trace, we replace the numeric value with a string `BINX` where `X` is the bin number in which the value is found. For example, if we have a variable for `agent_coal` and the values of this variable in the various traces are 15, 21, 23, 26, or 28, we might create the following histogram (bin label, range) pairs: (`BIN1`, [0-9]), (`BIN2`, [10-19]), (`BIN3`, [20-29]), and (`BIN4`, [30-∞]), such that `value(agent_coal, 21)` and `value(agent_coal, 23)` both become `value(agent_coal, BIN3)`. If we did not skolemize the numeric values, Word2Vec would interpret atoms that correspond to the same variable as completely different words when they have even slightly different numeric values. This would increase the vocabulary size of the corpus and make it more difficult to learn relationships concerning numeric atoms. By skolemizing, we ensure that values of the same state variable are recognized as the same word when they are within a certain range (i.e., the bin ranges). Previous similar work (Fine-Morris et al. 2020) skolemized all numerics to a single bin, but this erases all variation in numeric values, which is undesirable when trying to learn numeric landmarks.

Step 1.2: Learn Word Embeddings. We use Word2Vec with the skip-gram model to learn a set of word embeddings for the words in our corpus of traces. The properties of Word2Vec ensure that words that occur in the same context are clustered by cosine distance. A small cosine distance for two word embeddings indicates that the corresponding words frequently co-occur.

To learn word embeddings, we linearize traces such that each condition or action is treated as a word. For a trace $s_0, a_1, s_1, ..., s_N$, the linearized trace would be $c_{01}, ..., c_{0N}, a_1, c_{11}, ..., c_{1N}, ..., c_{N1}, ..., c_{NN}$, where all the conditions true in s_0 are represented by the set of conditions in the sub-sequence $c_{01}, ..., c_{0N}$ and each c is a condition true in state s_0. From a planning perspective, the ordering of the conditions within a state does not matter, and the conditions in a state can be reordered to create more input traces for Word2Vec.

Step 1.3: Select Landmark Atoms. The landmark learning process is as follows: we (1) learn word embeddings from the traces with Word2Vec, (2) form n=2 clusters of the word embeddings using a clustering algorithm and the cosine distance metric (we use Hierarchical Agglomerative Clustering), (3) score each atom according to the average cosine distance of the atom to those of the opposite cluster, (4) filter out non-effects atoms (i.e., any atom that isn't an effect of an action), (5) select atoms with scores less than $((max - min) \times .2) + min$ where max and min are the maximum and minimum scores of the effects atoms (i.e., any atom that is an effect of an action in at least one trace) and (6) replace all bin names with the associated numeric value range. The purpose of steps (2-3) are to split the atoms into a least two context groups and then find the atoms that are most in-between those two groups, i.e., the atoms in the corpus that has been assigned to one group but is as close as possible to another, possibly marking the boundary between the context groups. We choose two clusters semi-arbitrarily, as the best number of clusters will be domain-dependent and difficult to determine (we leave exploring this for future work).

4.2 Phase 2: Learn Methods

Phase two uses the landmark atoms to learn methods and consists of three steps. Step (2.1) partitions $\mathcal{T}$ using the landmark atoms, Step (2.2) learns subplan methods for each subtrace, and Step (2.3) learns landmark methods using the subplan methods.

In previous work (Fine-Morris et al. 2020), the final tasks of each trace were determined by finding which of a large set of user-provided possible final goals were present in the effects of the final action. In this work, each trace is annotated with a final task, which is not explicitly defined by a final goal condition, because this allowed tasks to be named more flexibly. Additionally, Fine-Morris et al. (2020) did not ground any of the numeric variables in the preconditions of the methods, while we selectively ground numeric variables to specific values when they are not already constrained by the calculations (otherwise values are constrained only by criteria inherited from actions).

During this phase, each trace is processed independently. Figure 1 shows how a hierarchy can be learned from one trace. The root contains the landmark method which has subtrees for each of its 4 subtasks: 1 for each of the three landmarks plus the final subtask method for final goal. The next level decomposes each landmark into a sequences of primitive tasks that accomplishes the landmark. All leaves in the tree are actions.

Step 2.1: Partition Traces. T2N accepts as input a set of traces, each annotated with the head of their final goal and a set of possible landmarks. We discard any traces which contain none of the landmarks because it is impossible to learn landmark methods from them. T2N partitions each trace, splitting at the states that contain the first instances of a landmark in the action effects. This results in a trace partitioned so that the last action of each subtrace achieves a landmark.

Step 2.2: Learn Subplan Methods. For each partition of the trace, T2N learns a method for accomplish the landmark that was accomplished by the subplan. The head of the subplan method comes from the landmark atom that occurs in the final state of the subtrace, modified so that any non-variable arguments are moved into the name. For example, a landmark such as `value(item, 1)` would become the method head `value_1(item)` because `item` denotes a variable but `1` does not.

We learn preconditions for a subplan method by performing goal regression using our modified regression technique over the subplan. We check to make sure that any variable that is updated in the subtrace and not already ground to a specific value in the preconditions have some calculation-based constraint applied to them in the precondition calculations. If not, we ground that variable according to its value in the initial state of the subtrace. We do this to ensure that variables which are unconstrained by the actions of the trace are still constrained by the context of the trace. We repeat this process for each remaining subtrace.

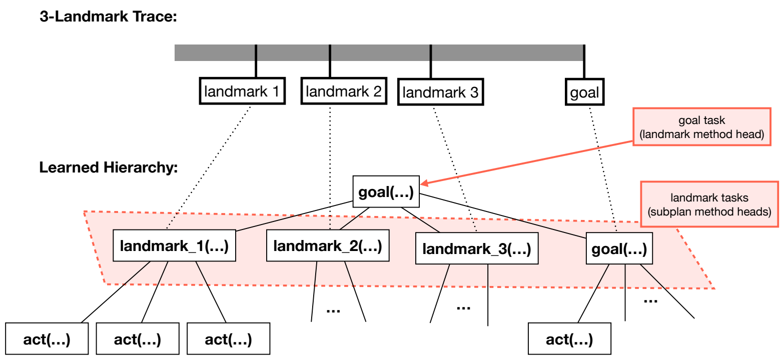

Figure 1: Example hierarchy showing the hierarchical structure learned from a trace with three landmarks. The structure learned from a trace is determined by the number of landmarks in the trace and the number of actions in each subtrace.

Each time we learn a method, we check to see if it can be merged with an existing method. Methods can be merged if they were learned from equivalent plans and if they are identical excepting their ground numeric variables. The merging process will ensure that any variables with numeric values will be transformed into a value range that includes both original values.

Step 2.3: Learn Landmark Method. Once we have methods for each (subtrace, subtask) pair, we can learn a landmark method. The head is the final trace goal, the subtasks are the landmarks plus the final trace goal. The preconditions are learned by via goal regression over the entire trace. As before, we ground any unconstrained numeric variables according to the first state of the trace.

An Example Hierarchy For a task such as `make stone_pickaxe(agent)`, the 3 landmarks of the example hierarchy of Figure 1 could be `value(agent_crafting_table, 1)`, `value(agent_wooden_pickaxe, 1)`, `value(agent_stone, 3)`, with `make stone_pickaxe(agent)` as the final goal. The head of the landmark method would be `make stone_pickaxe(agent)`, with 4 subtasks/subgoals: `value_1(agent_crafting_table)`, `value_1(agent_wooden_pickaxe)`, `value_3(agent_stone)`, `make stone_pickaxe(agent)`. The method for decomposing landmark_1 could contain a sequence of three primitive subtasks to gather wood, make planks, and make a crafting table.

5 Methods

All work for this paper was done on a 2020 MacBook Pro with a 2 GHz Quad-Core Intel Core i5 processor, 32 GB 3733 MHz LPDDR4X memory, running Big Sur (v. 11.6).

5.1 Training Problems and Solutions

To generate training traces, we first create randomized training problems, then use a planner to create solution plans (i.e., simulate plan demonstrations from a human). We use a customized version of the Pyhop planner with hand-crafted HTN methods, but as the plans are comprised of only state and action information and are not annotated with decomposition information, we believe that any planner will suffice. The training problems consist of initial states with counters for each gatherable item (wood, stone, iron, coal) indicating how many units of this resource are available for collection. The initial amounts of these counters were set randomly between 2000-5000 in steps of 100. Each agent (so far all traces have been single-agent) has a counter for each existing item type, all initialized to zero.

All tasks were to “make X” where X is some product/goal item that requires a crafting table to produce. The goal items were: [wooden, stone, iron] axe, [wooden, stone, iron] pickaxe, furnace, rail, or cart.

The hand-crafted methods were designed to produce plans with variability, not to produce the most efficient plans, or to find plans more quickly. We prioritized plan variability because we believe that this is more advantageous for the landmark learner, but we have yet to do a formal comparison. For each goal item we generate 10 plans. From a single set of plans we generate traces in the three different formats (i.e., FL, PE, and RA). We generate 20 traces from each plan by reordering the conditions in each state randomly. Each training trace is annotated with the head of the corresponding task to be learned.

5.2 Word2Vec Hyperparameters

We learn word embeddings using the following hyperparameters (we leave probing the impact of hyperparameters on landmark selection for future work). For all trace formats we used alpha=0.001, min alpha=0.0001, and 1000 epochs, with vector dimensions calculated according to to $V/20$, where V is the size of the corpus vocab, and window size is $3C$, where C is the average number of conditions per state. For PE traces, the vector size was therefore 9, and for RA and FL it was 11. The window sizes were 462, 78, and 48 for FL, RA, and PE, respectively.

5.3 Test Problems

We select test problems to examine the effectiveness of landmark methods. Landmark methods that were applicable to achieving specific goals were not learned for all goals in all method sets. Our test problems were selected to compare the performance of landmark methods, and by proxy the landmarks selected for each method set. If a particular landmark set results in learning a method set that doesn't cover certain goals, the failings of that set w.r.t. the other sets are obvious and don't require extensive testing.

Our selection procedure was designed with this in mind and is as follows: For each goal task, we generate a pool of 15 solvable test problems using the same technique for generating the training problems, but changing the starting ranges of the availiable gatherable items to a random number between 200-5000 in steps of 10. We confirmed that each problem was solvable by generating a solution plan using an HTN planner and the same hand-crafted methods used to generate the training traces. If the planner failed to generate a solution to test problem, we discarded and replaced the problem. Once we had a pool of solvable test problems for each goal task, we selected test problems via the following procedure for each method set: we (1) pool together all test problems for each goal task for which the method set has landmark methods and (2) select a set of 10 problems from the previously created multi-goal pools and add it to the set of selected methods. If a problem was selected for a previous method set, we selected a different problem (i.e. the final set of test problems should contain no duplicates).

5.4 Metrics

We used two metrics for comparing the four method sets: coverage and efficiency. **Coverage** measures how much of the domain is solvable by a particular method set, with subcomponents **goal coverage** and **problem coverage**. Goal coverage describes how many of the domain goals are address by at least one landmark method, regardless of how successful the problems with that goal are solved. If a method set has no landmark methods for a particular goal, it does not cover that goal. If a method set has landmark methods for a goal, but cannot solve any of the test problems with that goal in the time-limit, it still covers that goal. Problem coverage measures how successfully a method set solves test problems. **Efficiency**, the second metric, is concerned with how quickly a planner using a method set solves a particular test problem, or a set of test problems.

6 Results

Our results show that different ways of formatting the traces can produce very different sets of learned landmarks. We also find that landmark selection impacts which final goals are covered by a method set, and the efficiency with which plans for different goals are generated.

6.1 Landmarks

We learned three sets of landmarks (FL, PE, RA), one for each style of trace, and hand-selected a set of player-intuitive custom landmarks (CL) for a baseline (see Table 2). Of the landmarks learned by our algorithm, two were concerned with possessing a tool or station (wooden pickaxe in FL and furnace in RA), while most focused on gatherable resources (coal, iron ore, or cobblestone). The CL landmarks were more concerned with the possession of various types of pickaxes and two types of crafting stations (crafting table and furnace).

FL	value(agent_wooden_pickaxe, 1)
PE	value(cobblestone, [3896-4782])
	value(agent_cobblestone, 47)
RA	value(agent_coal, [10-18])
	value(agent_furnace, 1)
	value(agent_iron_ore, [10-18])
CL	value(agent_crafting_table, 1)
	value(agent_wooden_pickaxe, 1)
	value(agent_stone_pickaxe, 1)
	value(agent_furnace, 1)
	value(agent_iron_pickaxe, 1)

Table 2: All landmarks for each method set. Landmarks with a bin containing only a single value are specified with that discrete value, instead of the bin range (i.e., value(agent_crafting_table, 1)).

Goal- and Problem- Coverage. Both CL and FL method sets covered every goal in the set of test problems and demonstrated full problem coverage. Both RA and PE demonstrated full problem coverage for every covered goal, although neither covered the wooden_pickaxe goal and the former also did not cover the stone_pickaxe goal. In Table 3 we can see that only CL covers every final goal task. FL, PE, and RA are all missing landmark methods for "make wooden_axe". This is because none of them have landmarks pertinent to the task, so no landmark methods could be learned.

Goal Tasks	CL	FL	PE	RA
make cart	10	10	1	10
make furnace	7	7	3	7
make iron_axe	10	10	2	10
make iron_pickaxe	10	10	1	10
make rail	9	9	3	10
make stone_axe	8	8	3	2
make stone_pickaxe	6	6	3	0
make wooden_axe	5	0	0	0
make wooden_pickaxe	4	4	0	0

Table 3: Number of landmark methods in each method set for each task. Goals with a 0 are not covered by the method set of that column.

From these results, we can see that the formatting of the trace provided to Word2Vec to generate embeddings impacts the atoms selected as landmarks. This can impact the landmark methods the system can learn, because traces which contain none of the selected landmarks have to be discarded. If all traces demonstrating how to achieve a particular end goal are discarded, we cannot learn that goal.

6.2 Planning Duration

Figure 2 shows the amount of time the planner took to solve each test problem. Problems occur in the same order for all 4

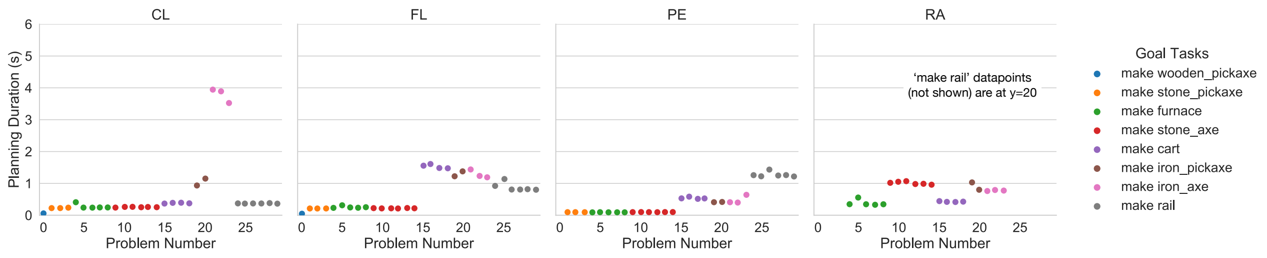

Figure 2: Number of seconds to find a correct plan using the four sets of learned methods.

graphs, and are grouped by goal (see legend). No method set showed obviously superior performance across all goal categories, however, the landmarks used during method learning can clearly have an impact on planning efficiency, as some of the method sets solve problems of a certain category faster than others.

In general, when problems are solved with the same set of methods, there is not much difference in the solving time of problems with the same goal. All attempted problems were solved in less than 30 seconds, but problems were only attempted if the goal of the problem was covered by a landmark method in the method set.

A primary purpose of these experiments was to determine how changes to the formatting of traces impacted the learning of landmark atoms, and therefore the learning of methods. We also sought to determine if learning methods and using them to solve problems would provide us with information on the utility of the learned landmarks, as it is otherwise difficult to judge the effectiveness of the landmark learning technique, including the Word2Vec hyperparameters, the best formatting for the traces input to Word2Vec, and the technique for selecting landmarks using word-embeddings.

It is difficult to declare any one set of method better. Each had their strengths and weaknesses. Problems solved with the CL method set were mostly completed in less than a half second. Goal categories that were outliers included those with iron_pickaxe and iron_axe goal products, which rose to about a second and about four seconds respectively. Notably, it was much more efficient on the rail problems than any other method sets.

For the PE method set, the solvable lower-cost goals (stone_pickaxe, furnace, stone_axe) were accomplished slightly more quickly than for any other method set, mostlikely because the landmarks were all concerning cobblestone, an ingredient of stone axes/pickaxes and the furnace. The uptick in planning time for iron products is probably related to the slightly greater difficulty of deciding how much coal and iron to gather, and how much iron to smelt. Rails require the most iron, so they had the greatest time cost.

For FL, duration for the low-cost items are roughly the same as for CL and PE, but for the cart goal the duration jumps up by more than a second to nearly 1.5 seconds, and then decreases gradually for the remaining goals (iron_pickaxe, iron_axe, rail) stopping at slightly less than a second. This almost inverts the trend seen in the PE graph for the iron goal products, where rail is more expensive than cart. (Note that, because there is only one landmark atom selected for FL, the landmark methods can have only two subtasks: one for making the wooden pickaxe and one for making the final product.)

In some goal categories, RA performed worse than all other method sets (furnace and stone_axe) and demonstrated middle-of-the-road performance for other categories (cart, iron_pickaxe, iron_axe) and failed to cover both stone_pickaxe and wooden_pickaxe. Additionally, its performance on the rail goal was an extreme outlier at about 20 seconds while solutions for all other problems and method sets were achieved in less than 5 seconds. This is unexpected as the RA landmarks, which involve the furnace, iron_ore, and coal, should be useful for learning efficient solutions.

The comparatively poor performance on furnace and stone_axe is probably due to unnecessary resource gathering; the RA landmarks target resources that are not needed for either goal product, and therefore methods are only learned from traces for those goals when they do something inefficient (for example, like crafting a stone and iron_pickaxe in order to gather stone to make a stone_axe).

7 Related Work

Fine-Morris et al. (2020) uses a similar technique to learn methods with numeric preconditions, leveraging Word2Vec and clustering techniques. It then uses these landmarks to partition the traces and learn a similar hierarchy of methods with numeric preconditions of arbitrary complexity. The crucial difference is that T2N allows for goals to be numeric fluents. This enables the agent to solve problems where, for example, there is a minimum number of resources needed.

Word2HTN (Gopalakrishnan, Munoz-Avila, and Kuter 2018) uses a technique involving Word2Vec and clustering similar to the one described in this work, with similar inputs and outputs. A significant difference between our system and Word2HTN is that Word2HTN learns an exclusively binary hierarchy (every method has two subtasks) as opposed to our system, where the top-level methods decompose the task into arbitrarily-many subtasks, capturing complex underlying task structures beyond right-recursive task decompositions. Additionally, while Word2HTN can learn numeric preconditions and effects involving arithmetic operations only (e.g., addition, subtraction), our system can learn more complex (i.e., compound) functions via function composition. This has trade-offs, as Word2HTN can simplify its numeric expressions in ways our system cannot (for example, it can combine two updates to $varX$, $+3$ and -2 into one $+1$ update. Because of this, the preconditions learned by our system can be very large.

Both HTN-Maker (Hogg, Muñoz-Avila, and Kuter 2016) and HTNLearn (Zhuo, Muñoz-Avila, and Yang 2014) learn task decomposition methods from traces and user-provided annotated task definitions comprised of (precondition, effects) pairs. HTNLearn uses constraint satisfaction to learn method preconditions, while HTN-Maker uses goal regression and can learn right-recursive subtasks. Crucially, in our work we are not giving the subtasks as input; while we annotate our traces with the goal-task, we identify the bounds of a task using the occurrence of learned landmarks in the effects of an action and did not require extensive user-provided subtask specifications. Also, neither handles numeric fluents.

Segura-Muros et al. (2015) learns HTN planning domains from plan traces using a combination of process mining and inductive learning. Process mining builds a behavioral model from a set of event logs. By treating plan traces as event logs the authors use process mining to learn the hierarchical structure of the plan traces. Once the structure is learned, they extract pre-state and post-state pairs from the plan traces for each action and method and utilize inductive learning to generate preconditions and effects. They show that their algorithm can learn a simple and straightforward domain capable of consistently solving test problems. However, they do not handle numerics.

ICARUS (Langley, Choi, and Rogers 2007) uses background knowledge and means-ends analysis to learn Teleoreactive logic programs from provided solution plans. The background knowledge includes concept definitions, which are composed together to form more complex concepts. The hierarchy of concepts this creates helps define the hierarchical structure of the programs. Since tasks are linked to the achievement of concepts which are built upon achieving goals, Teleoreactive logic programs basically encode HGNs. Our system requires less user-provided domain information to dictate hierarchical structure, instead inferring structure from the learned landmarks.

X-learn (Reddy and Tadepalli 1997) learns goal-subgoal relations. X-learn uses inductive learning to learn d-rules with preconditions and a sequence of fully-ground subgoals (similar to HGN decomposition methods or macro-actions) from increasingly-difficult exercises. X-learn is purely symbolic, unlike our system which can learn numeric goals.

8 Final Remarks

We have discussed our results from learning several sets of HTN methods using an automated learner, T2N, for a domain with numeric goals. We tested them on 30 test problems and discussed how the set of landmarks used to structure the learned methods can have an impact on which goals that method set covers. No consistent pattern exists in the efficiency of methods for all goals across method sets, but the set of landmarks impacts the efficiency with which different method sets solve problems for the same goal. Our results indicate that the selected landmarks can have an impact on the final goals that are covered by a method set, and that it is possible that different trace formats can impact landmark selection.

While it is difficult to draw firmer conclusions about the impact of trace formatting without more data, results thus far suggest that some method sets are complementary in terms of the problems and goals they solve. Although method sets may fall short of complete domain coverage, it is possible that the weakness of each could be remedied by using parallel planners, one for each method set. By taking the plan of the first planner to finish, we could cover the complete set of goal tasks. While this would be more computationally expensive, it would take advantage of the strengths of both learned method sets. We suggest this as an alternative to merging method sets, which might prove challenging to do without degrading performance.

The work presented here is preliminary, with possible future research including: (1) comparing the performance of landmarks learned using multiple bins with those learned from a single bin, (2) using clustering of numerics for assigning bin labels instead of discretely-sized bins, (3) performing the same experiments with different Word2Vec hyperparameters, (4) experimenting with more than 2 clusters during bridge atom selection, (5) trying to replicate the results in other domains, and (6) finding ways to prevent goals from going uncovered by landmark methods during learning due to landmark-less traces being skipped, for example, by introducing the ability to select fallback landmarks for landmark-less traces.

Acknowledgments. We thank NRL and ONR for funding this research and NSF's Independent Research and Development (IR/D) Plan. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

Bundy, A.; and Wallen, L. 1984. Skolemization. In Bundy, A.; and Wallen, L., eds., *Catalogue of Artificial Intelligence Tools*, 123–123. Berlin, Heidelberg: Springer. ISBN 978-3-642-96868-6.

Fine-Morris, M.; Auslander, B.; Floyd, M. W.; Pennisi, G.; Muñoz-Avila, H.; and Gupta, K. M. 2020. Learning Hierarchical Task Networks with Landmarks and Numeric Fluents by Combining Symbolic and Numeric Regression. In *Proceedings of the 8th Annual Conference on Advances in Cognitive Systems*, 16.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Elsevier. ISBN 978-0-08-049051-9. Google-Books-ID: uYnpze57MSgC.

Gopalakrishnan, S.; Munoz-Avila, H.; and Kuter, U. 2018. Learning Task Hierarchies Using Statistical Semantics and Goal Reasoning. *AI Communications*, 31(2): 167–180.

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

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2016. Learning Hierarchical Task Models from Input Traces.

Computational Intelligence, 32(1): 3–48. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/coin.12044.

Langley, P.; Choi, D.; and Rogers, S. 2007. Interleaving Learning, Problem Solving, and Execution in the Icarus Architecture. 27.

Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G. S.; and Dean, J. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc.

Nau, D.; Muñoz-Avila, H.; Cao, Y.; Lotem, A.; and Mitchell, S. 2001. Total-Order Planning with Partially Ordered Subtasks. In *Proceedings of the 17th international joint conference on Artificial intelligence - Volume 1*, IJCAI'01, 425–430. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 978-1-55860-812-2.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2014. On the Extraction, Ordering, and Usage of Landmarks in Planning. In *Sixth European Conference on Planning*.

Reddy, C.; and Tadepalli, P. 1997. Learning Goal-Decomposition Rules using Exercises. In *Proceedings of the fourteenth national conference on artificial intelligence and ninth conference on Innovative applications of artificial intelligence.*, 843–843.

Segura-Muros, J. A. 2015. Learning HTN Domains using Process Mining and Data Mining techniques. *Workshop on Generalized Planning (ICAPS-17)*, 8.

Shivashankar, V.; Kuter, U.; Nau, D.; and Alford, R. 2012. A Hierarchical Goal-Based Formalism and Algorithm for Single-Agent Planning. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*, 9. Valencia, Spain.

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