# Automated Learning of Hierarchical Task Networks for Controlling Minecraft Agents

Chanh "Sam" Nguyen Noah Reifsnyder Sriram Gopalakrishnan Hector Munoz-Avila

Department of Computer Science & Engineering; Lehigh University; Bethlehem, PA 18015

**Abstract. In this paper, we present an agent that learns Hierarchical Task Network (HTN) knowledge from observing a player performing actions in Minecraft. From these observations, the agent learns the tasks that the player pursues and how to achieve these tasks. We present an HTN learning algorithm and report on experiments of the agent assisting a player performing tasks in Minecraft.**

## I. Introduction

Hierarchical-Task Networks (HTNs) are an expressive planning knowledge formalism (Tate 1977). In HTN planning, high-level tasks are recursively decomposed into simpler ones until primitive tasks or actions are generated. These actions are the ones executed by an agent. HTNs consist of four classes of planning knowledge artifacts (exemplified on a gaming environment): (1) atoms indicating true conditions in the world (e.g., avatar is next to a tree) or desired conditions, called goals (e.g., 3 units of wood need to be harvested); (2) operators indicating actions that an avatar can take (e.g., harvest wood); (3) tasks indicating activities that can be performed by the agents (e.g., construct an axe); and (4) methods indicating how to achieve the tasks (e.g., to construct an axe, the agent needs to find a tree, move next to it, harvest it and use the resulting wood chunks to build an axe).

HTN planning has been shown to be strictly more expressive than STRIPS planning (Erol et al, 1994); loosely speaking, we can represent planning knowledge using the 4 knowledge artifacts indicated that cannot be represented when using only atoms and operators (i.e., the first two knowledge artifacts). An intuitive example is expressing the problem of an agent patrolling a circuit of 5 locations starting and ending in the same location an arbitrary number of times (e.g., the kind of behavior that would be expected in a NPC guard in a game such as a Thief).

A number of game researchers and practitioners have pointed to the importance of HTN planning for controlling NPCs (see related work). Those works assume someone has engineered the HTN knowledge. In practice, however, crafting HTN knowledge is known to be an intensive knowledge engineering task (Hogg et al., 2008). The challenge is that methods are basically recipes on how to combine actions to achieve tasks (an action is an instance of an operator). Thus, a knowledge engineer must not only craft what individual actions an agent might take but also craft high-level tasks and the means to combine actions and simpler tasks to achieve those more complex tasks.

While several studies have been done on learning HTN knowledge (see related work), these have focused on synthetic domains. In this paper, we present our ongoing work on learning HTNs tailored specifically for gaming environments such as Minecraft. Our work assumes as input:

- Action traces of Minecraft players performing unknown tasks
- The operators generalizing individual actions that can be taken by these players
- The partial states observed before and after the actions are taken

We developed software infrastructure on top of Microsoft's project Malmo that enables us to log actions taken by players while playing Minecraft. Our infrastructure is independent of the player being controlled by a person or by a program. We parse the log and, using the operators, automatically generate training traces consisting of actions taken by the player and the intermediate states observed between those actions. From these our agent is able to automatically learn the following knowledge artifacts:

- Tasks indicating high-level activities performed by the agents as extracted from the traces
- Methods for accomplishing the tasks as extracted from the traces

We also report on experiments assessing the capabilities of our learning agent. The paper continues as follows: the next section presents preliminaries formalizing the notion of HTN planning using Minecraft as an example. Then, we present our agent's framework, the HTN learning algorithm and how the agent performs assigned tasks. Next, we present an empirical evaluation of our agent, followed by a discussion of related work. Finally, we discuss future work.

## II. Preliminaries

The agent knows which actions to execute by following plans. A **plan** $\pi = <a1\ a2\ \ldots\ an>$ is a sequence of actions. **Actions** are instances of operators. An **operator** $o = (h\ prec\ neff\ peff)$ is a 4-tuple where $h$ is the head of the operator, *prec* are the list of preconditions, *neff* are the list of negative effects and *peff* are the list of positive effects. We refer to the operator's effects to indicate both positive and negative effects. The head of the operator is represented as an atom: $(name\ arg_1\ \ldots\ arg_m)$ where *name* is a string and $arg_1, \ldots, arg_m$ can be either constants or variables. The preconditions, negative effects, and positive effects are each a list of atoms. The following is an example of an operator in the Minecraft:

**Operator Name:** CraftIronPickaxe (Agent_1)
**Operator Preconditions:**
(Agent_1 hasIronIngots >=3, Agent_1 hasSticks >=2)
**Effects**:

(Agent_1 hasIronIngots -3, Agent_1 hasSticks -2, Agent_1 hasIronPickaxe +1)

This operator has *Agent_1* (a Minecraft agent) crafting an iron pickaxe. The preconditions are: (1) The agent has at least 3 iron ingots, and (2) the agent has at least 2 sticks. The effects are: (1) the agent has one more iron pickaxe, (2) 3 fewer iron ingots, and (3) 2 fewer sticks. This operator has no negative effects.

When the arguments of an atom are all constants, the atom is said to be grounded. An action is a grounded instance of an operator: that is all variables in the arguments of the head, preconditions, and effects have been instantiated to constants. Actions indicate how the states are transformed. A state is represented as a list of grounded atoms. An action $a = (h\ prec\ neff\ peff)$ is **applicable** to an state $S$ if $prec \subseteq S$. Applicability conditions take into account numerical conditions, checking if there is enough of the required materials in the state. Applying $a$ to $S$ results in the state $S' = (S - neff) \cup peff$. Numerical conditions are applied by making arithmetic transformations in the resulting state. Given a starting state S, and a plan $<a_1\ a_2\ \dots\ a_n>$, a **plan transformation** is a sequence: $\pi = <S\ a_1\ S_1\ a_2\ S_2\ \dots\ a_n\ S_n>$ such that $a_1$ is applicable to S and $S_1$ is the result of applying $a_1$ to S, $a_2$ is applicable to $S_1$ and $S_2$ is the result of applying $a_2$ to $S_1$, and so forth.

Our Minecraft agent generates plans using HTN planning. In HTN planning complex tasks are recursively decomposed into simpler tasks, until primitive tasks are reached. Tasks are represented as atoms. There are two kinds of tasks in HTN planning: **compound tasks**, which are tasks that can be decomposed into simpler tasks, and **primitive tasks**, which are achieved by actions. Given a primitive task $t$ and a state $S$, an action $a$ achieves $t$ if $a$ is applicable to $S$ and $t$ is the head of $a$. Applying a transforms the state S as defined before.

Compound tasks are decomposed by methods. A **method** $m = (h\ prec\ ST)$ is a triple where $h$ is the head of the method, *prec* are the list of preconditions, and *ST* are the list of tasks, referred to as the **subtasks** of m. The following is an example of a method that our agent learns Minecraft:

**Method Task:** Achieve-goal-1(Agent_1 hasLogs MATH~+1~)
**Method Preconditions**:
(Location_1 contains wood, Location_1 X xVal1, Location_1 Y yVal1, Location_1 Z zVal1,
Agent_1 X xVal2, Agent_1 Y yVal2, Location_1 Z zVal2)
**Method Subtasks:**
I. Achieve-goal-1(Agent_1 X xVal1, Agent_1 Y yVal1, Location_1 Z zVal1)
II. Achieve-goal-2(Agent_1 hasLogs MATH~+1~)

This method shows how the Minecraft agent can increase the number of logs (wood) it has. The precondition is to find a location that has wood (in Minecraft these are 3-dimensional locations *(xVal1, yVal1, zVal1)*). The subtasks are to (I) go to that location, and (II) harvest the wood. The second subtask differs from the method's head task because the name difference will direct it to a different method, which decomposes the task into a primitive action to "HarvestLog(Agent_1 Location_1)"

Given a compound task $t$ and an state $S$, a method $m = (h\ prec\ ST)$ decomposes $t$ if there exists a substitution $\phi: V \rightarrow C$ (where $V$ is all variables in $m$ and $C$ is all constants in the domain) such that: $\phi(h) = t$ and $\phi(prec) \subseteq S$. Like with the actions, the applicability conditions are extended to take into account numerical conditions. The **decomposition** of $t$ using m is $\phi(ST)$.

An **HTN planning problem** is defined as a 4-tuple *(O, M, S, T)*, where $O$ is a collection of operators, $M$ is a collection of methods, $S$ is a state (i.e., the starting state) and $T$ is a list of tasks (i.e., the tasks that we want to achieve). HTN planning recursively decomposes tasks in T until a plan is generated $\pi = <a_1\ a_2\ \dots\ a_n>$.

Given $O$, a collection of plan transformations $\Pi$ and $T'$, the collection of all heads of operators in $O$, our agent learns $M$ and $T$, with $T' \subseteq T$. The tasks learned have the form *(t, pre, peff)* where $t$ is a compound task, *pre* are a set of preconditions and *peff* are positive effects. The tasks learned in this way tell the player what their semantics are so that the player knows which task to apply as needed.

## III. A Framework for HTN Learning in Virtual Environments

We envision agents acting as companions of players performing tasks in virtual environments. The agent learns how to perform tasks by observing the player acting on the environment. Our agent makes the following assumptions and has the following capabilities:

- The agent parses actions from the player's execution traces.
- The agent knows neither the tasks that the player is trying to achieve nor are the traces annotated with the specific tasks or subtasks that the player is trying to achieve.
- The agent knows the semantics of the actions, namely (task, preconditions, effects) triples.
- The traces include state information (e.g., location of resources, where the agent moves to, etc.)

We implemented these ideas using Microsoft's Project Malmo. We implemented an automated player built by domain experts to perform a variety of tasks in the domain. The learning agent has no direct access to the expertise of the automated player. Instead, as the player takes actions in the domain these are executed using the Malmo interface. The execution is recorded in a trace. The agent parses the traces to generate plan transformations. The following shows an example of a trace generated by Malmo (the first bracket is a

player' action, followed by a time stamp and parameters. The second bracket is the state):

*[Crafting log- planks [u 'log', 1)], 2017-03-09 00:01:52.386689-x=36 y=45 z=59-[Log-1, planks-1, stick-0, wooded_pickaxe-0…*

Traces consist of hundreds such action-parameter-state triples. Such traces are generated regardless if the player is automated or a human player. So henceforth we refer to the automated player as the player. The agent parses these traces to identify actions taken by the player. For example, the trace typically contains sequences of low-level moving actions and then the player proceeds to make a series of wood harvesting actions collecting 3 pieces of wood. The agent parses these as "MoveTo(wood), Harvest(Wood,3)", which form part of a plan transformation.

## IV. An HTN Learning Algorithm

Our agent uses the plan transformations to learn tasks and methods. The following is the pseudocode for the agent's HTN Learning process:

**HTNLearning** (Π,O, T)
\\ **input**: *Π* a collection of plan transformation; *O* a collection of operators; *T* the collection of primitive tasks achieved by operators in *O*
**Output**: *M* a collection of methods, *T'* a collection of compound tasks decomposed by the methods in *M:*
*1. M ← {}*
*2. Parse Π to generate a vector representation $V_{\Pi}$ of plan elements in Π*
*3. Use clustering techniques to generate a hierarchy H of the plan elements in $V_{\Pi}$*
*4. Recursively identify landmarks T' in H*
*5. for each t ∈ T' from the bottom to the top*
    *5.1 for each decomposition of t in H*
    *construct a method m*
        *5.1.1 M ← M ∪ {m}*
*6. Return (T', M)*

In Step 2 we use Word2Vector (Le & Mikolov, 2014) for learning semantic word embeddings from sentences. In our work we view a plan transformation $\pi$ = <*S a1 S1 a2 S2 … an Sn*> as a sentence; whereby each action $a_i$ and each atom in the states are viewed as words. We use Word2Vector to produce vector representations of these atoms and actions. Word2Vector uses shallow neural networks consisting of one hidden layer. These neural networks are trained to represent the semantic relationships between atoms and actions by using the contexts in which the actions and atoms occur in the traces. As a result, the vector representations for the actions and states' atoms are dependent on the contexts in which they occur; the vector representations for states' atoms and actions that frequently co-occur will have closer vector representations than those that co-occur infrequently. The similarity between two embeddings is measured by the dot product of their normalized vectors (i.e., the Cosine distance). So relative orientation defines similarity rather than the magnitude of the vectors.

In Steps 3 and 4 we perform a Hierarchical Agglomerative Clustering (HAC) (Ward, 1963) on the vector representations and extract the landmarks. For our work, we performed 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 groups were merged. The result of the HAC process is a clustering matrix representing how the words are grouped hierarchically. From the HAC matrix, we can determine the landmarks for a specified set of words (the vocabulary of the traces). A **landmark** is the word in the last grouping that has the highest sum (total) of the similarity with other words. We remove this and repeat the process getting an ordered set of landmarks. These landmarks are the tasks T' we are learning.

In Step 5, we observe all possible ways that the landmark t was decomposed in H and learn one method for each such decomposition. The basic idea is as follows: we start with landmarks whose subtasks are actions (the "bottom" of the hierarchy). We refer to these landmarks as level-1 tasks; their children are primitive tasks (we refer to these as level-0 tasks). We learn a decomposition of (level-1) task t into primitive tasks *ST* (all level-0 tasks) and collect the preconditions *prec* of these actions to learn a method *m = (t prec ST)*. For each subsequent level *k* (with k ≥ 2), a level-k task *t* may have as children tasks *ST* of level *k-1*,*k-2*, .., *0*. For example, a level-2 task *t* may have as children *ST* a combination of level-1 and level-0 tasks. We collect the preconditions *prec* of these actions and tasks to learn a method *m = (t prec ST)*.

## V. A Taskable Agent

We use the learned HTN domain (O, M, S, T) (T includes the primitive and compound tasks) to create a taskable agent. This agent is designed to assist the player in performing the tasks. The player knows the tasks that the agent is capable of performing by looking at its learned task semantics (t, pre, peff). For example one of the tasks the agent learned in Minecraft is:

1. Task: furnace-task
2. Preconditions: ()
3. Effects: has(furnace, 1)

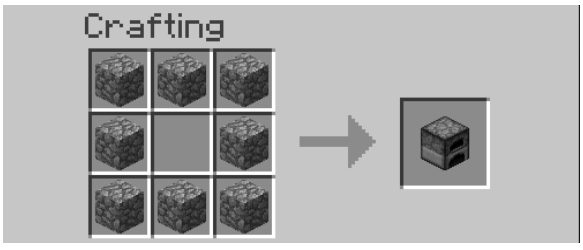


***Figure 1**: Snapshot of the recipe to create a furnace*

This task has no requirements and when it is completed the agent will have created a furnace. In Minecraft, this is a long procedure requiring the agent to locate and harvest enough pieces of wood to produce a wooden pickaxe. With this pickaxe, the agent locates and harvests enough stones and then it can create the furnace (Figure 1).

The player never tells the agent what tasks the player is performing. But from its observations, the agent infers the tasks and the ways to achieve these tasks. Furthermore, since the agent is learning by observing the player, it mimics the ways the player performs these tasks. This can be important to increase the player's trust in the agent.

The agent uses an HTN planner to generate a plan based on the player-selected task t, the current state S and the HTN domain learned. Actions in this plan are executed in Malmo using scripts. For example, the action MoveTo(wood) triggers a script that locates a nearby wood from the agent and moves to that location.

## VI. Empirical Evualuation

The traces used to train the agent were generated from an (automated) player constructing iron pickaxes. In our experiments, the baseline is the time it takes for the player to build 2 iron pickaxes. We test the time it takes to construct 2 iron pickaxes when the player builds one pickaxe and the agent is tasked with building the other one.

We build our Minecraft environment with three simplifications. First, we don't require a crafting table to craft items (Malmo platform allows this). However, to add more complexity, we still require our agent to have a furnace to craft iron ingots from iron ores. Second, we use the FlatWorld generator provided by Malmo to simplify navigation tasks for both our automated player and the learning agent. The FlatWorld generator generates a flat environment, where the agents, as the name of the generator suggests, only need to navigate in the x and z-direction and not the y-direction (which is the up and down direction in Minecraft). Third, cobblestones and iron ores in our agent's world are placed in columns similar to how wooden blocks are placed in trees consisting of 4 blocks (Figure 2). This is different from the typical settings in Minecraft game, where cobblestones and iron ores are often found underground instead. As our primary concern is the mining and crafting actions by the agent and not on low-level navigation tasks, these three are reasonable simplifications. In such an environment, we task our agent with creating iron pickaxes within Minecraft. This task requires many subtasks, thereby showing the benefit from Hierarchal Task Network plan representations. This is specifically true because the subtasks are not all ordered. We developed a scenario with targeted changes to allow for controlled testing as well. The world size is limited by a boundary, and resources, as well as the agent, are randomly placed inside this boundary. The agent must collect the appropriate materials and craft the correct tools to allow it to achieve the goal of creating an iron pickaxe. For example, to mine cobblestones, the agent has to be holding a wooden pickaxe (or better-quality pickaxes).


***Figure 2:*** *The player and the agent*

A sample sequence of player's actions would look something like this: Move to Wood, Collect Wood, Craft Wooden Pickaxe, Move to Cobblestone, Collect Cobblestone, Craft Stone Pickaxe, Craft Furnace, Move to Iron Ore, Collect Iron Ore, Craft Iron, Craft Iron Pickaxe. The states between actions that call for the next action relies on the agent's current inventory. For instance, the Goal at the start is Craft Iron Pickaxe. This has the dependencies of (Has Wood, Has Iron)

**The automated player**. It uses the observations available through the Malmo platform. Since it is able to see and analyze the environment, it can make decisions to accomplish its goals. The goals in this instance are to craft some items. In order to craft the required item, the player recursively follows the fixed recipes for the items in Minecraft. The order of the required ingredients in a recipe is then randomized at the beginning of each run to generate a better variety in the produced plan traces**.** The actions that the player can perform are a bank of low-level functions to execute the menial tasks of moving, breaking a block, and crafting an item.

**The agent**. The learned agent has access to the same bank of low-level functions as the player. The difference is that the agent is given a plan generated by the HTN planner to carry out. There are 3 types of actions in the plans: MoveTo, Harvest, and Craft. MoveTo takes an argument, which is the type of the resource (material) that the agent should move to nearby. For example, "MoveTo: log" means that the agent should move next to the closest wood block according to its observation of the environment. Similarly, "Harvest: log", assuming that the agent is already next to a wood block, will have the agent hit the wood block in front of it and once the block breaks, go gather the log drop. Lastly, "Craft" will have the agent craft an item, provided that it has the necessary ingredients in its inventory. The following is an excerpt from a plan produced by the HTN planner to craft an iron pickaxe from scratch: *{MoveTo: log, Harvest: log, Craft: planks, MoveTo: log, Craft: stick}*

Accordingly, executing this excerpt, the agent will gather a log, craft wooden planks, move to a log, craft a stick, move a log and harvest it. Interestingly, one would expect a

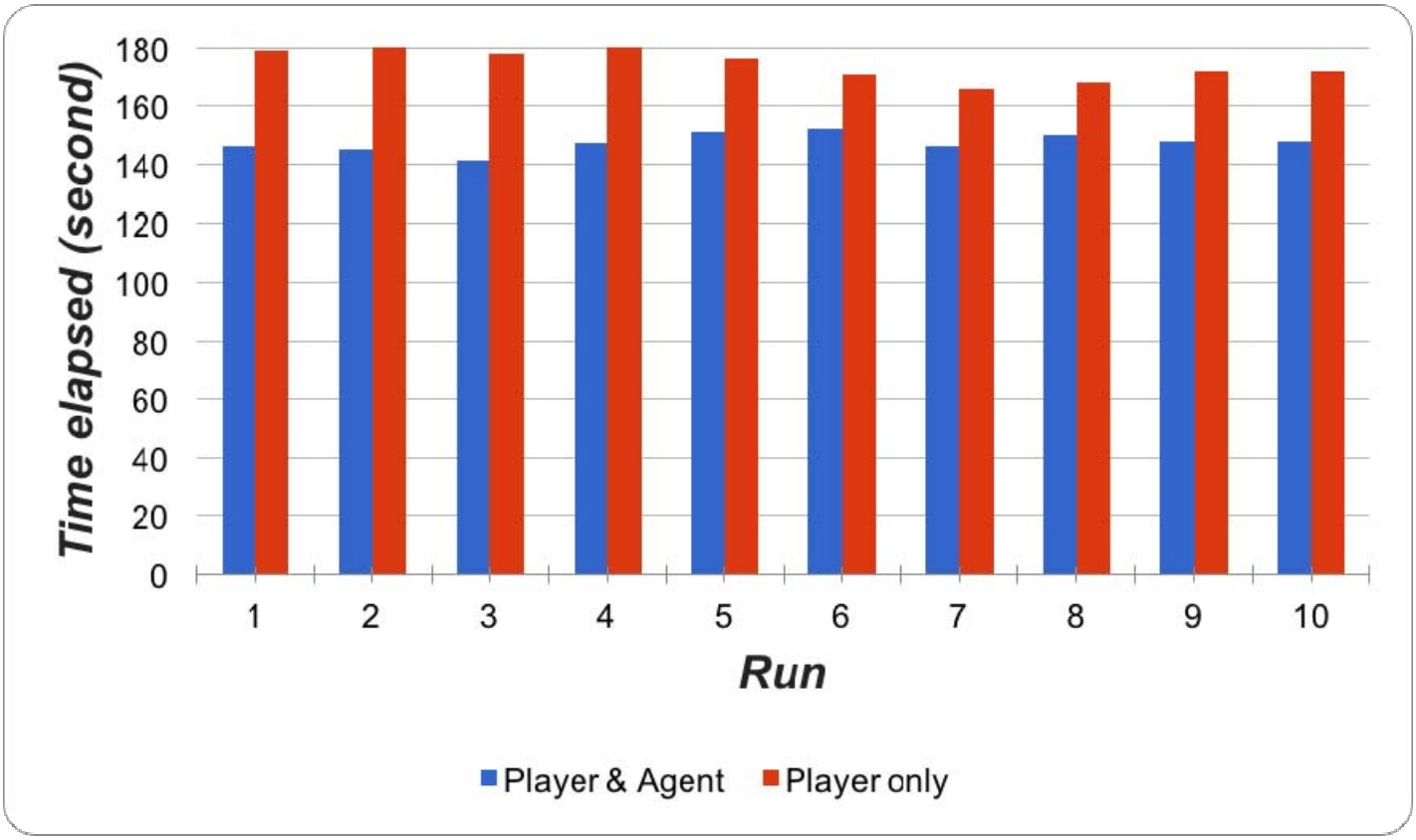


***Figure 3:*** *Individual Runs vs. Time to complete Tasks*

"Harvest: log" after the second "MoveTo: log". Instead, it's a "Craft: stick". This is because we randomize the order of the iron pickaxe recipe for each run of the player that produces the plan traces, from which the HTN planner learns. Thus, as a result, the produced plan can have some variance in its order.

We set up two different scenarios to test if the plan produced by the HTN planner works effectively. For the first one, we tasked the player with crafting 2 iron pickaxes. We noted the time it took and repeated this for 10 runs. The resources (wood, stone and iron ore), as well as the player, were randomly placed at the beginning of each run so the order of the player's actions and paths may differ from one run to the next. For the second scenario, we paired the player with the agent (after training) and tasked each one with making an iron pickaxe. Again, we noted the time until both of them finished the task and repeated the scenario 10 times.

The results are presented in Figure 3. Player & Agent each tasked with making an iron pickaxe has an average runtime of 147.4 seconds and a median runtime of 147.5 seconds whereas those numbers for the player making 2 iron pickaxes are 174.2 seconds and 174 seconds, respectively. Player & Agent finished making 2 pickaxes 16% faster than where only the human was making the items. This shows that the plan produced by the HTN planner is effective and it can be of value to a player when utilized.

## VII. Related Work

A number of research has been reported on the use of HTN planning techniques for games. Ontanon and Buro (2015) combines HTN planning with game tree for creating effective automated players in Real-Time Strategy games. These players take into account an opponent's potential actions and generate a plan accordingly. Smith et al. (1998) also reasons about an opponent's possible moves and combines it with HTN planning but for playing bridge, a turn-based card game. Meijer and Koppelaar (2001) explore reasoning on an adversary's moves during HTN planning for playing Go.

Researchers have pointed that HTN's representation capabilities refining high-level strategies into more concrete ones are particularly useful for creating robust NPC behavior (Hoang et al., 2005). Kelly et al. (2007) describes an architecture using HTN planning to generate scripts controlling NPCs and test it in the RPG game Oblivion (from Bethesda's Elder Scrolls Series).

These ideas are extended to control teams of NPCs (Gorniak and Davis, 2007). Straatman et al. (2013) reports on the deployment of HTN planning techniques to control bots in the commercial game Killzone 3, an FPS game. Soemers and Winands (2016) report on the use of HTN plans in FPS games; it uses case-based reasoning techniques to store and reuse the plans without requiring the action model.

Learning HTNs has two components: (1) learning the hierarchical decompositions relating tasks and subtasks and (2) learning the applicability conditions for this decomposition. Some works focus on learning the applicability conditions assuming the hierarchical decompositions are given as input (Ilghami et al., 2005; Xu and Munoz-Avila, 2005). However, most existing works learn the applicability conditions and the task hierarchies. These works elicit a hierarchy from a collection of plans and from a given action model (Choi et al. 2005, Hogg et al., 2008, Zhuo et al., 2014). Those works assume that the task semantics are given as input and have been tested on synthetic domains only.

## VIII. Conclusions and Future Work

We present an agent capable of learning HTN planning knowledge from traces automatically generated from observing players accomplishing Minecraft's tasks. The agent learns the tasks that the players are pursuing as well as the HTN methods to accomplish these tasks. The tasks tell the player what kinds of activities the agent can perform. This capability of agents to perform a variety of tasks is called taskability, the capability of agents to learn how to perform tasks from humans (Laird, 2014). Our agent uses the learned knowledge to assist the player. We tested our agent in Minecraft as an assistant of an automated player. Combined the player and the agent performed tasks faster than the player performing the tasks by itself. Crucially, the agent only knew of the basic actions that could be performed; what tasks to achieve and how to achieve them was learn entirely from the traces.

In future work, we want to explore using the learned knowledge to learn HTN planning knowledge that takes into account quality considerations. For example, the agent might want to minimize the amount of time it takes to accomplish its assigned tasks and might ponder about the different ways it could accomplish these tasks. Other quality considerations might include minimizing the materials needed to constructing the required tools.

**Acknowledgements**. This material is work supported in part by the National Science Foundation and the Office of Naval Research.

## References

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

[2] Erol, K., Hendler, J., & Nau, D. S. (1994). HTN planning: Complexity and expressivity. In Proceedings of AAAI-94.

[3] Ontanón, S., & Buro, M. (2015, July). Adversarial hierarchical-task network planning for complex real-time games. In Proceedings of the 24th International Conference on Artificial Intelligence. AAAI Press.

[4] Gorniak, P., & Davis, I. (2007). SquadSmart: Hierarchical Planning and Coordinated Plan Execution for Squads of Characters. In AIIDE-2007.

[5] Hoang, H., Lee-Urban, S., & Muñoz-Avila, H. (2005). Hierarchical Plan Representations for Encoding Strategic Game AI. In AIIDE 2005.

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

[7] Humphreys, T. (2013) Exploring HTN Planners through Examples. in Game AI Pro: Collected Wisdom of Game AI Professionals, 1st ed., S. Rabin, Ed. CRC Press, ch. 12.

[8] 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.

[9] Kelly, J. P., Botea, A., & Koenig, S. (2007). Planning with hierarchical task networks in video games. In Proceedings of the ICAPS-07 Workshop on Planning in Games.

[10] Laird, J. (2014) Report on the NSF-funded Workshop on Taskability. Technical report. University of Michigan.

[11] Le, Q. V., & Mikolov, T. (2014). Distributed Representations of Sentences and Documents. In ICML-14.

[12] Meijer, A. B., & Koppelaar, H. (2001). Pursuing abstract goals in the game of Go. In Belgium-Netherlands Conference on AI (BNAIC).

[13] Soemers, D. J., & Winands, M. H. (2016). Hierarchical Task Network Plan Reuse for video games. In Computational Intelligence and Games (CIG), 2016 IEEE Conference on Games. IEEE.

[14] Smith, S. J., Nau, D. S., & Throop, T. A. (1998). Success in spades: Using AI planning techniques to win the world championship of computer bridge. AAAI/IAAI, 98, 1079-1086.

[15] Straatman, R., Verweij, T., Champandard, A., Morcus, R., & Kleve, H. (2013). Hierarchical AI for multiplayer bots in Killzone 3. Game AI Pro.

[16] Tate, A. (1977). Generating project networks. In Proceedings of the IJCAI-77. Morgan Kaufmann Publishers Inc.

[17] Ward Jr, J. H. (1963). Hierarchical grouping to optimize an objective function. Journal of the American statistical association.

[18] Xu, K., & Muñoz-Avila, H. (2005). A domain-independent system for case-based task decomposition without domain theories. In AAAI-05.

[19] Zhuo, H. H., Hu, D. H., Hogg, C., Yang, Q., & Munoz-Avila, H. (2009). Learning HTN Method Preconditions and Action Models from Partial Observations. In IJCAI-2009.