Visualization of an Autonomous Minecraft Agent Trained with Hierarchical Task Networks

Elizabeth Koshelev

Department of Computer Science and Engineering

Lehigh University Smart Spaces REU

Last update August 3 2017

1 Introduction

This project, hosted at the Smart Spaces REU program at Lehigh University, consists of visualizing an autonomous, artificially intelligent Minecraft agent that has been trained with hierarchical task network learning techniques. The project was implemented in Project Malmo, a platform built on Minecraft used for artificial intelligence research and experimentation. The procedure for utilizing hierarchical task networks is fully outlined in “Automated Learning of Hierarchical Task Networks for Controlling Minecraft Agents" (Nguyen, Reifsnyder, Gopalakrishnan, Munoz-Avila). In this paper we begin by discussing the development environment, Project Malmo, and how to use it. We then discuss how the agent was trained through hierarchical task network structures. Finally, we discuss how the agent observes its environment and navigates back to the player. The cycle of the demo is as follows-the user tells the agent to make a pickaxe, the agent makes it and returns to the user, and then the user can give the agent new recipes to execute. The product of this project can be found by following this link: https://www.youtube.com/watch?v=NQe25BEfeSQ.

2 Development and Test Environment

In this section we discuss the installation of Project Malmo and the structure of a Malmo Project. Each Malmo project consists of two parts—the XML file describing the environment, and the code describing the mission that the agent will take.

2.1 Installation

Installation for Malmo, with details, can be found at https://github.com/Microsoft/malmo. Make sure that all dependencies are configured, and the location of Malmo and python are in a path with no spaces. On Windows, make sure to use Windows PowerShell and not the command line. After installed, test if everything was configured correctly by running an instance (for example, on windows Powershell navigate to cd C:\Users\Public\Malmo\Minecraft and execute .\launchClient.bat). Then, because our project is in python, we navigate to cd C:\Users\Public\Malmo\Python_Examples and execute python tutorial_1.py. If all goes well, the world of Minecraft should load up with a sample mission.

2.2 XML

The XML file describes the Minecraft world and the agent, as well as the observations the agent can take from the environment. It can be split into two sections—the server section, and the agent handler section. The XML part appears as pictured below.

```
missionXML='''<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
            <Mission xmlns="http://ProjectMalmo.microsoft.com" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

              <About>
                <Summary>Hello world!</Summary>
              </About>

              <ServerSection>
                <ServerHandlers>
                  <FlatWorldGenerator generatorString="3;7,220*1,5*3,2;3;,biome_1"/>
                  <ServerQuitFromTimeUp timeLimitMs="30000"/>
                  <ServerQuitWhenAnyAgentFinishes/>
                </ServerHandlers>
              </ServerSection>

              <AgentSection mode="Survival">
                <Name>MalmoTutorialBot</Name>
                <AgentStart/>
                <AgentHandlers>
                  <ObservationFromFullStats/>
                  <ContinuousMovementCommands turnSpeedDegs="180"/>
                </AgentHandlers>
              </AgentSection>
            </Mission>'''
```

Server Initialization

The server section of the XML file describes the environment. One can change the environment, starting variables, and time limit on the mission in this section. To control the actual world, a generator string can be used. To create one you must use an external website such as http://chunkbase.com/apps/superflatgenerator. You can also use the function DrawingDecorator to hardcode elements in the environment. This is pictured below.

```
            <DrawingDecorator>

                <DrawCuboid type="bedrock" x1="-1" y1="0" z1="-1" x2="''' + str(worldSize + 1) + '''" y2="''' + str(
                stoneHeight + 34) + '''" z2="''' + str(worldSize + 1) + '''"/>
                <DrawCuboid type="air"  x1="0" y1="1" z1="0" x2="''' + str(worldSize) + '''" y2="''' + str(
                stoneHeight + 34) + '''" z2="''' + str(worldSize) + '''"/>
            <DrawCuboid type="stone" x1="0" y1="1" z1="0" x2="''' + str(worldSize) + '''" y2="''' + str(
                stoneHeight) + '''" z2="''' + str(worldSize) + '''" />
            <DrawCuboid type="dirt" x1="0" y1="''' + str(stoneHeight + 1) + '''" z1="0" x2="''' + str(
                worldSize) + '''" y2="''' + str(stoneHeight + 3) + '''" z2="''' + str(worldSize) + '''" />
            <DrawCuboid type="grass" x1="0" y1="''' + str(stoneHeight + 4) + '''" z1="0" x2="''' + str(
                worldSize) + '''" y2="''' + str(stoneHeight + 4) + '''" z2="''' + str(worldSize) + '''" />

                ''' + getTrees() + '''
                ''' + getCobbleStones() + '''
                ''' + getIronTrees() + '''
                ''' + getCoals() + '''

            </DrawingDecorator>
```

To understanding what commands can be used to shape the environment, please reference the tutorial pdf found in the Python_Examples folder.

Agent Handler

The second part of the XML file initializes an agent. For an agent to receive specific observations and execute certain actions, specific permissions must be granted in the AgentHandlers section. For example, if the agent needs to observe the chat you must include <ObservationfFromChat /> and if you want the agent to take commands in chat you must include <ChatCommands />. There are two types of movement in Malmo: continuous and discrete. To allow either type it must be specified in this section. For example, below is the XML code to initialize the Minecraft agent in our project.

```
<AgentHandlers>
    <ObservationFromDistance>
        <Marker name="End" ''' + getEnd() + '''/>
        <Marker name="Zero" x="0" y="0" z="0"/>
    </ObservationFromDistance>
    <ObservationFromChat />
    <ObservationFromFullStats/>
    <ObservationFromRecentCommands />
    <ObservationFromNearbyEntities>
        <Range name="nearby" xrange="10" yrange="10" zrange="10" update_frequency="1" />
    </ObservationFromNearbyEntities>
    <ObservationFromFullInventory/>
    <ObservationFromRay/>
    <ObservationFromGrid>
    ''' + getObservationGrid() + '''
    <Grid name="Blocks">
      <min x="-7" y="-7" z="-7"/>
      <max x="7" y="7" z="7"/>
    </Grid>
    </ObservationFromGrid>
    <ContinuousMovementCommands turnSpeedDegs="180">
        <ModifierList type="deny-list">
            <command>strafe</command>
        </ModifierList>
    </ContinuousMovementCommands>
    <SimpleCraftCommands/>
    <InventoryCommands/>
    <ChatCommands />
</AgentHandlers>
```

2.3 Initializing the Mission

The second part of the Malmo project is the code to initialize the mission and the agent. The basic layout begins with necessary imports. It is imperative to include MalmoPython as the code will not run without it.

```
import MalmoPython
import os
import sys
import time
import json
import math
```

Next, there is a line to initialize the agent and mission, as well as catch any errors in creating the Malmo objects.

```
agent_host = MalmoPython.AgentHost()
try:
    agent_host.parse( sys.argv )
except RuntimeError as e:
    print 'ERROR:',e
    print agent_host.getUsage()
    exit(1)
if agent_host.receivedArgument("help"):
    print agent_host.getUsage()
    exit(0)

my_mission = MalmoPython.MissionSpec(missionXML, True)
my_mission_record = MalmoPython.MissionRecordSpec()
my_mission.observeRecentCommands()
```

Then, the mission is initialized and lines are written to catch any exceptions and allow for a certain number of retries should the mission fail to begin.

```
max_retries = 3
for retry in range(max_retries):
    try:
        agent_host.startMission( my_mission, my_mission_record )
        break
    except RuntimeError as e:
        if retry == max_retries - 1:
            print "Error starting mission:",e
            exit(1)
        else:
            time.sleep(2)
```

Finally, there are lines to initialize the agent's world_state perspective, and the code to start the mission.

```
world_state = agent_host.getWorldState()
while world_state.is_mission_running:
    world_state = agent_host.getWorldState()
    #Here you give the agent commands
    sys.stdout.write(".")
    time.sleep(1)
    for error in world_state.errors:
        print "Error:",error.text
```

These sections can be seen in their simplest form in tutorial_1. In general, the tutorials are immensely helpful and it is highly recommended to go through the Tutorial.pdf to get a good introduction to Malmo.

2.4 Controlling the Agent

In Malmo, there are many ways to control the agent. You can move forward, turn side to side, and pitch up and down. The agent can take many observations such as the grid in its line of sight, commands from the chat, and see its inventory.

Movement Commands

Basic movement commands are implemented using the format agent_host.sendCommand("move 1") in the while loop that starts the mission. For example:

```
while world_state.is_mission_running:
    time.sleep(0.1)
    world_state = agent_host.getWorldState()
    agent_host.sendCommand("move 1")
    for error in world_state.errors:
        print "Error:",error.text
```

Discrete movement commands move the agent a certain distance, and continuous movement commands move the agent continuously until it receives a different command. In our project we utilize continuous commands, because there were problems with using both discrete and continuous in the same project for reasons unknown. Here is a list of sample commands:

```
agent_host.sendCommand("move 1")
agent_host.sendCommand("turn -1")
agent_host.sendCommand("pitch 1")
agent_host.sendCommand("move 0")
agent_host.sendCommand("chat Hello world!")
agent_host.sendCommand("discardCurrentItem")
```

Using time.sleep(x) makes the agent sleep for x seconds before processing a new command, and is extremely useful for refreshing observations and moving only a certain distance using continuous commands.

Observations

Observations are stored in a JSON file that must be called every time you would like to receive an observation. If you want a specific observation from the environment, make sure that it is specified in the AgentHandler section of the XML file. Here is an example of how an agent called Sudo would take information from the environment, and then use a specific chat command to make a pickaxe.

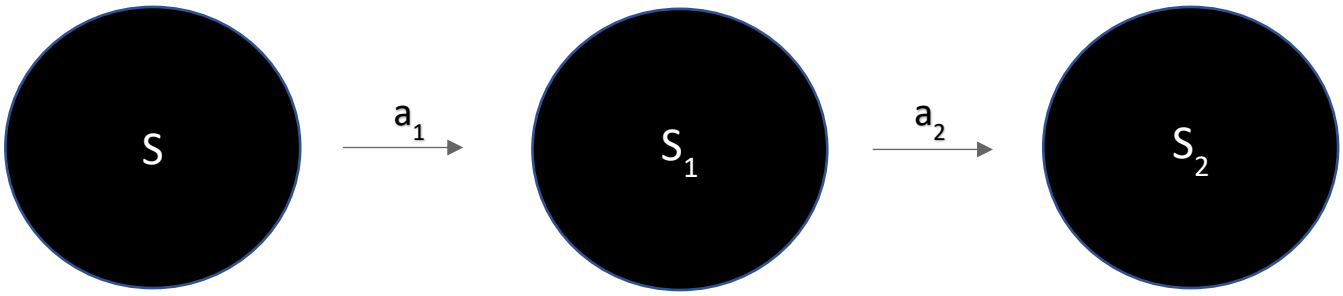
```
def observeWorld():
    while agent_host.peekWorldState().is_mission_running:
        for obs in agent_host.peekWorldState().observations:
            msg = obs.text
            ob = json.loads(msg)
            chat = ob.get(u'Chat', "")
            x = ob.get(u'XPos')
            z = ob.get(u'ZPos')
            y = ob.get(u'YPos')
            pitch = ob.get(u'Pitch')
            yaw = ob.get(u'Yaw')
            grid = ob.get(u'LineOfSight')
            for command in chat:
                parts = command.split("> ")
                if "Sudo make" in parts[1]:
                    if "stone_pickaxe" in parts[1]:
                        action.writeOriginalPickaxetoTraces()
                        makeItem("stone_pickaxe")
```

3 Theory

Hierarchical task networks, (abbreviated HTNs) are powerful planning architectures that take a complex task as input, and recursively break it down into a series of simple tasks. In this section we discuss how the agent utilizes HTNs to make a pickaxe based on observations from a player. For more information please reference the original paper: "Automated Learning of Hierarchical Task Networks for Controlling Minecraft Agents" (Nguyen, Reifsnyder, Gopalakrishnan, Munoz-Avila) CIG-2017.

3.1 HTN Terms

- **Compound tasks** are tasks that can be decomposed into simpler tasks, implemented using methods.
- **Primitive tasks** are achieved by actions.
- 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 subtasks (i.e. to construct an axe, agent needs to locate a tree walk there harvest it and craft an axe).
- Given a starting state S, and a plan <a1 a2 … an>, a **plan transformation** is a sequence: π = <S a1 S1 a2 S2 … an Sn>.



- An **atom** is a truth or desired condition (called a goal) of the world i.e Goal: 3 units of ore need to be harvested, Truth: Avatar is next to a tree
- An **operator** is an action an avatar can take (i.e. harvest wood).
- A **task** is an activity that can be performed (i.e. construct a plank).

3.2 Training the Agent

The agent is trained through traces of the actions of an automated player. An example of a trace is as such:

> [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…]

The agent doesn't know the tasks or subtasks that the player is trying to achieve when it observes the traces, but it knows the semantics (task, preconditions, effects) of the traces. 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. The training process can be broken down into 5 steps, as demonstrated by this diagram from "Automated Learning of Hierarchical Task Networks for Controlling Minecraft Agents" (Nguyen, Reifsnyder, Gopalakrishnan, Munoz-Avila) CIG-2017.

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 $V_Π$ of plan elements in Π
3. Use clustering techniques to generate a hierarchy H of the plan elements in $V_Π$
4. Recursively identify landmarks T' in H
5. for each t ∈ T' from the bottom to the top
5.1for each decomposition of t in H construct a method m
5.1.1 M ← M ∪ {m}
6. Return (T', M)

In step 1, the data structure to hold the methods is initialized. In step two, a program called Word2Vector (Le & Mikolov, 2014) that utilizes shallow neural networks for sentence decomposition is used to convert a plan transformation π = <S a1 S1 a2 S2 … an Sn> to a vector representation V_r. In step 3, Hierarchical Agglomerative Clustering (HAC) (Ward, 1963) to generate a hierarchy of plan elements from V_r Then, landmarks in V_r are found by recursively breaking down the hierarchy. A **landmark** u is the atom in the grouping of (U,V) that is used to break V down into tasks A trace is split into goals achieved before the landmark, and the goals achieved after the landmark. Observing the landmarks starting from the bottom of the hierarchy, we may identify primitive tasks and their parents, as well as their preconditions and we may learn a method m = (task preconditions primitive tasks).

4 Method

In this section we discuss how the agent receives commands from the player in game and executes them. First, we talk about how the agent observes the chat to figure out what item to make. Next, we talk about how it executes the "traces.txt" file to make the item. Finally, we discuss how the agent navigates back to the player and drops the item after it makes it. The action cycle is pictured below.

4.1 Sending Commands

To send commands to the agent, we have it observe the chat, and break each observation into parts— the first part specifies the person who sent the chat, and the second part is the chat itself. To observe the command that tells Sudo to make the pickaxe, we take the line with "Sudo make" and execute the traces to create that item. The syntax to pick up different observations is pictured below.

```
def observeWorld():
    while agent_host.peekWorldState().is_mission_running:
        for obs in agent_host.peekWorldState().observations:
            msg = obs.text
            ob = json.loads(msg)
            chat = ob.get(u'Chat', "")
            x = ob.get(u'XPos')
            z = ob.get(u'ZPos')
            y = ob.get(u'YPos')
            pitch = ob.get(u'Pitch')
            yaw = ob.get(u'Yaw')
            grid = ob.get(u'LineOfSight')
            for command in chat:
                parts = command.split("> ")
                if "Sudo make" in parts[1]:
                    if "stone_pickaxe" in parts[1]:
                        action.writeOriginalPickaxetoTraces()
                        makeItem("stone_pickaxe")
```

4.2 Executing Traces

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. A command such as MoveTo: wood triggers a script found in “traces.txt” that locates wood based on the HTN learning techniques. Here is an example of the “traces.txt” file for making a wooden pickaxe:

```
MoveTo: log
Harvest: log
Craft: planks
MoveTo: log
Craft: stick
MoveTo: log
Harvest: log
Craft: planks
MoveTo: cobblestone
Craft: wooden_pickaxe
```

To execute these traces, there are many different files used to perform different commands. For example, the MalmoCraftUtilities file contains the commands to execute traces, and is called from the file MalmoTraceUtilities. A good example of this is the definition gather_basic_block which combines many different commands from another file, MalmoMineUtilities to gather a basic ingredient. The MalmoGlobals files is used to load in the traces, as follows:

```
def load_traces():
    traces = []
    with open("traces.txt") as traces_file:
        content = traces_file.readlines()
        for t in content:
            tokens = t.split(":")
            kind = tokens[0].strip()
            item = tokens[1].strip()
            trace = Trace(kind, item)
            traces.append(trace)
        print traces
        return traces
```

MalmoLogUtilities is used to print different observations to aid in the coding process. For example, here is a definition used to return a timestamp:

```
def get_time_second():
    dt = datetime.fromtimestamp(time.time()).strftime('%Y-%m-%d %H:%M:%S.%f')
    return dt
```

Finally, the MalmoConstants file is full of constants used to identify traces as such (where C is the MalmoConstants file):

```
if trace.kind == C.KIND_HARVEST or trace.kind == C.KIND_MOVE_TO:
```

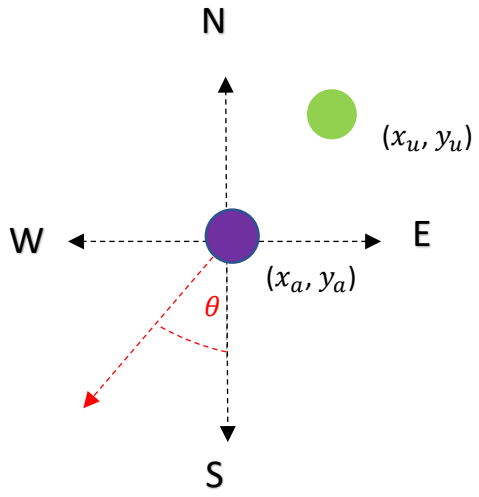
The traces are executed one by one with the following command, which in turn calls several of the different files above to execute the trace.

```
done = MTU.process_trace(trace, agent_host, observations, ax, ay, az, a_yaw, a_pitch, grid3d,
                         nearby_entities,
                          line_of_sight, G.current_block_to_hit, G.extra_data)
```

4.3 Moving back to the user

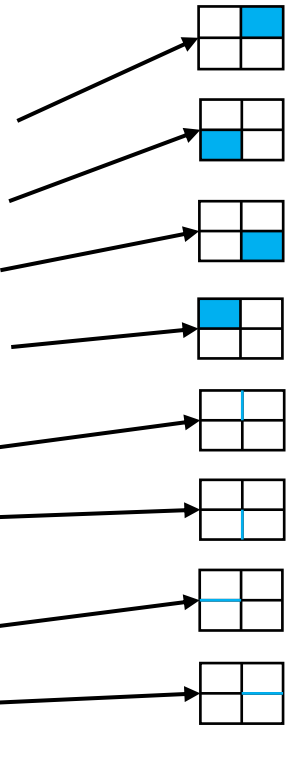
After the traces are executed and the item is finished, the agent must navigate back to the user. Because we use continuous commands, to navigate back to the user we must find the angle that the agent must face to walk forward and reach the player. In Malmo, a polar coordinate system is used— South is 0°,West is 90°,North is 180°,East is 270°. Yaw refers to the direction the agent is facing with respect to these angles, and pitch is how much the agent is looking up and down. As a note, the yaw can be both negative and above 360 °. In the diagram below, we have an eagle view of the Minecraft, with the agent being represented by the green dot and the player being represented with the purple dot. Thus, we use the equation $yaw = tan^{-1}\frac{|dy|}{|dx|}$ to find the

angle that the agent must go, and then adjust it based on the quadrant that the player is in relative to the player.



With the blue squares/lines being the quadrant the player is in relative to the agent, the pseudocode is pictured below.

```
findYaw(xa,xu, ya, yu)
    dy = ya − yu
    dx = xa − xu
    if dx!=0:
        yaw = tan^-1 |dy|/|dx|
    if dx<0 and dy<0:
        yaw=270+yaw
    if dx<0 and dy>0:
        yaw=270-yaw
    if dx>0 and dy<0:
        yaw=90-yaw
    if dx>0 and dy>0:
        yaw=90+yaw
    if dx=0:
        if dy>0:
            yaw=180
        if dy<0
            yaw=0
    if dy=0
        if dx>0:
            yaw=90
        if dx<0
            yaw=270
```



4.4 Creating the database

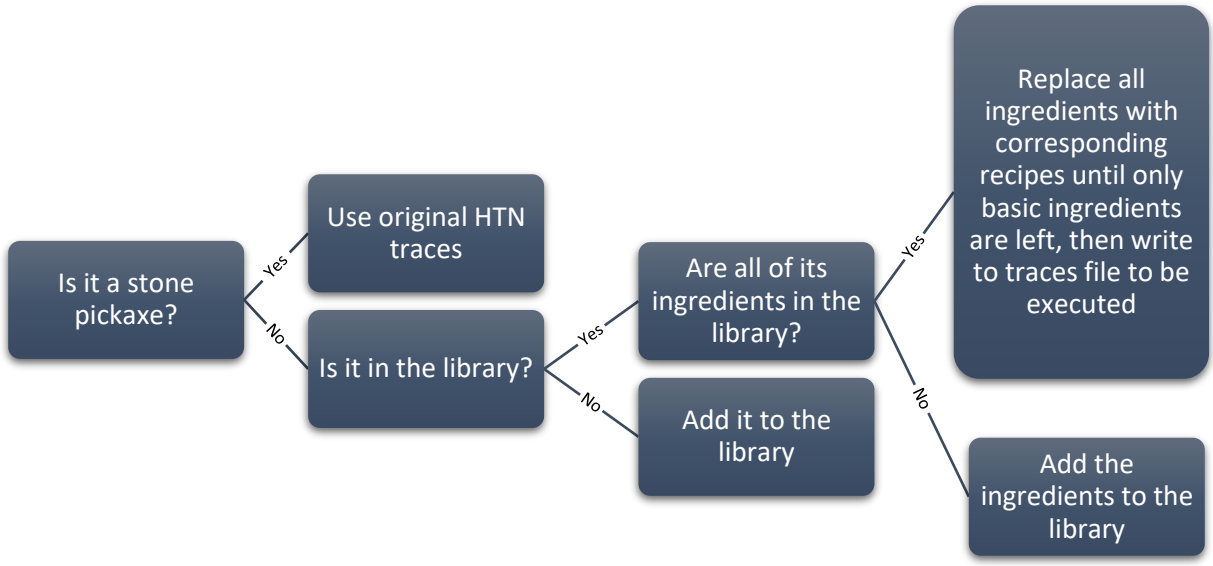
To create the database, we may add new recipes to text file “library” in format “IngredientsforX: Y Z”. Thus, the format of the library looks like this:

```
Ingredientsforwooden_pickaxe: planks planks planks stick stick
Ingredientsforstick: planks planks
```

We check for if the item already exists in the database, and if its ingredients exist as well by iterating through the library and creating a list of item lists in the form of an array of arrays, then checking the corresponding items. If an item is not found, the agent will state that this is the case.

4.5 Creating traces from the database

The process for creating traces from the library is as follows: if making the original pickaxe, copy the HTN traces from the original project. Otherwise, we check if the item’s recipe is in the library and if all of its ingredients exist in the library as well. We then make an array of the ingredients, and take all those that are not basic ingredients (such as a log, iron_ore etc.) and replace them with their recipes until there are only basic ingredients in the trace. Then, we replace all basic resources with their corresponding trace. For example, we replace log with MoveTo: log Harvest: log. The final array is written into the traces file and executed just as the pickaxe trace was. The decision tree for how to use the agent to create items is as follows:



5 Conclusion

Because Minecraft has such a unique environment, Project Malmo is a powerful tool for artificial intelligence research and experimentation. In this project we demonstrate the power of using hierarchical task networks to train an agent to create a pickaxe. We discuss the code, theory, and platform behind the demo as well as the paper that inspired it. We create a cycle of giving the agent tasks, adding to its library, and having it navigate back to the player. In doing so, a we create a library of recipes that the agent can execute. This provides a useful artificially intelligent assistant to Minecraft players.

Future Work

Currently, the agent depends heavily on the user inputting new recipes in the right form. Future work could have the agent try and check if the recipes are correct, and then adjust them if they are not. As well as this, future work may involve learning from demonstration, where the agent observes the player performing a task and then can replicate it in real time. As well as this, the agent could learn recipes itself through trial and error. Another important factor to consider is that in this demo, the agent was put in a controlled enclosed environment. Putting the agent in a full Minecraft environment would be the next step, though there would be a lot of limiting factors to consider, such as getting caught in a hole or a cave and being able to recognize and run from monsters in the game. As well as this, the agent cannot currently build anything, only craft. In the future, a building command could be implemented for the agent for a more diverse set of skills. Overall, there is a great amount of potential for machine learning in Minecraft, especially when using a platform such as Project Malmo.

References

Nguyen C., Reifsnyder N., Gopalakrishnan S., Munoz-Avila H. (2017) Automated Learning of Hierarchical Task Networks for Controlling Minecraft Agents. In the proceedings of the Conference on Computational Intelligence in Games.

Johnson M., Hofmann K., Hutton T., Bignell D. (2016) The Malmo Platform for Artificial Intelligence Experimentation. Proc. 25th International Joint Conference on Artificial Intelligence, Ed. Kambhampati S., p. 4246. AAAI Press, Palo Alto, California USA. https://github.com/Microsoft/malmo

Le, Q. V., & Mikolov, T. (2014). Distributed Representations of Sentences and Documents. In the proceedings of the International Conference on Machine Learning.

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

This material is based upon work supported by the National Science Foundation under Grant numbers IIS-1217888 and IIS-1359280.