A Implementation Details

A.1 Action Abstraction

Instruction for Extracting Structured Actions. To extract structured actions, we first ask LLM to generate a tree-structured action planning for each of the 3141 predefined tasks provided by MineDojo, and then converts each action step into a `(verb, object, tool, material)` tuple. During decomposition, it is essential to ensure actions are neither too broad nor too specific. We adjusted the depth of the action decomposition tree to achieve balance, and empirically set the depth as 2 to meet our requirements.

Specifically, we use `gpt-3.5-turbo` from OpenAI API to generate the structured actions. We add the following instruction to the content of "SYSTEM" role to generate the tree-structured plan. We add the goal description, *e.g.*, "find material and craft a iron pickaxe", to the content of "USER" role and then asks LLM to response according to the requirements.

> **SYSTEM:**
> You serve as an assistant that helps me play Minecraft.
>
> I will give you my goal in the game, please break it down as a tree-structure plan to achieve this goal.
>
> The requirements of the tree-structure plan are:
>
> 1. The plan tree should be exactly of depth 2.
> 2. Describe each step in one line.
> 3. You should index the two levels like '1.', '1.1.', '1.2.', '2.', '2.1.', etc.
> 4. The sub-goals at the bottom level should be basic actions so that I can easily execute them in the game.
>
> **USER:**
> The goal is to {`goal description`}. Generate the plan according to the requirements.

After that, we extract the action tuple from each sentence of the leaf nodes. We use the following instruction as the content of "SYSTEM" role to extract the tuple and add the sentence to the content of "USER" role.

> **SYSTEM:**
> You serve as an assistant that helps me play Minecraft.
>
> I will give you a sentence. Please convert this sentence into one or several actions according to the following instructions.
>
> Each action should be a tuple of four items, written in the form ('verb', 'object', 'tools', 'materials')
>
> 'verb' is the verb of this action.
> 'object' refers to the target object of the action.
> 'tools' specifies the tools required for the action.
> 'material' specifies the materials required for the action.
> If some of the items are not required, set them to be 'None'.
>
> **USER:**
> The sentence is {`sentence`}. Generate the action tuple according to the requirements.

Then, we extract the structured actions by selecting frequent actions and merging actions with similar functionalities. The set of structured actions is {`equip`, `explore`, `approach`, `mine/attack`, `dig_down`, `go_up`, `build`, `craft/smelt`, `apply`}. Note that we disregard more detailed action decomposition for `attack` and `build` to remove overly detailed short-term actions and focus on long-term task completion.

A.2 LLM-BASED HIERARCHICAL AGENT

A.2.1 LLM DECOMPOSER

We use `gpt-3.5-turbo` from OpenAI API [2] for goal decomposition. The prompt is shown as follows, which consists of two parts: instruction with the role of "SYSTEM" and query with the role of "USER". The {`object quantity`}, {`object name`} and {`related knowledge`} are injectable slots that will be replaced with corresponding texts before fed into the LLM.

SYSTEM:

You are an assistant for the game Minecraft.

I will give you some target objects and some knowledge related to the object. Please write the obtaining of the object as a goal in the standard form.

The standard form of the goal is as follows:
{
"object": "the name of the target object",
"count": "the target quantity",
"material": "the materials required for this goal, a dictionary in the form {material_name: material_quantity}. If no material is required, set it to None",
"tool": "the tool used for this goal. If multiple tools can be used for this goal, only write the most basic one. If no tool is required, set it to None",
"info": "the knowledge related to this goal"
}

The information I will give you:
Target object: the name and the quantity of the target object
Knowledge: some knowledge related to the object.

Requirements:
1. You must generate the goal based on the provided knowledge instead of purely depending on your own knowledge.
2. The "info" should be as compact as possible, at most 3 sentences. The knowledge I give you may be raw texts from Wiki documents. Please extract and summarize important information instead of directly copying all the texts.

Goal Example:
{
"object": "iron_ore",
"count": 1,
"material": None,
"tool": "stone_pickaxe",
"info": "iron ore is obtained by mining iron ore. iron ore is most found in level 53. iron ore can only be mined with a stone pickaxe or better; using a wooden or gold pickaxe will yield nothing."
}
{
"object": "wooden_pickaxe",
"count": 1,
"material": {"planks": 3, "stick": 2},
"tool": "crafting_table",
"info": "wooden pickaxe can be crafted with 3 planks and 2 stick as the material and crafting table as the tool."
}

USER:

[2] `https://platform.openai.com/docs/api-reference`

Target object: {object quantity} {object name}
Knowledge: {related knowledge}

The recursive decomposition generates a sub-goal tree starting from the final goal object as the root node: if a goal has some prerequisites (materials or tools), for each required material or tool, we add a child node representing the goal of obtaining that material or tool, and then recursively decompose the child node, until there is no more prerequisites. The related knowledge is from: 1) Crafting/smelting recipes in MineDojo (Fan et al., 2022), written in the form "Crafting {quantity} {object} requires {material} as the material and {tool} as the tool"; 2) Wiki on the Internet [3]. We extract the paragraphs with keywords "obtaining", "mining", "sources", etc.

A.3 LLM PLANNER

Here we present the prompt for planning with LLM. We also use `gpt-3.5-turbo` from OpenAI API as the LLM planner. The model accepts inputs in the form of a chat, i.e., the prompt is a dialogue consisting of several messages, each of which contains a role and the content. We set the `Instruction` with the role "SYSTEM" at the beginning, and use the `User Query` with the role "USER" to query the LLM for response. The content of the `Instruction` and `User Query` are as follows.

A.3.1 INSTRUCTION

SYSTEM:

You serve as an assistant that helps me play the game Minecraft.

I will give you a goal in the game. Please think of a plan to achieve the goal, and then write a sequence of actions to realize the plan. The requirements and instructions are as follows:

1. You can only use the following functions. Don't make plans purely based on your experience, think about how to use these functions.

`explore(object, strategy)`
Move around to find the object with the strategy: used to find objects including block items and entities. This action is finished once the object is visible (maybe at a distance).
Augments:
- object: a string, the object to explore.
- strategy: a string, the strategy for exploration.

`approach(object)`
Move close to a visible object: used to approach the object you want to attack or mine. It may fail if the target object is not accessible.
Augments:
- object: a string, the object to approach.

`craft(object, materials, tool)`
Craft the object with the materials and tool: used for crafting new object that is not in the inventory or is not enough. The required materials must be in the inventory and will be consumed, and the newly crafted objects will be added to the inventory. The tools like the crafting table and furnace should be in the inventory and this action will directly use them. Don't try to place or approach the crafting table or furnace, you will get failed since this action does not support using tools placed on the ground. You don't need to collect the items after crafting. If the quantity you require is more than a unit, this action will craft the objects one unit by one unit. If the materials run out halfway through, this action will stop, and you will only get part of the objects you want that have been crafted.
Augments:
- object: a dict, whose key is the name of the object and value is the object quantity.

[3] https://minecraft-archive.fandom.com/wiki/Minecraft_Wiki

- materials: a dict, whose keys are the names of the materials and values are the quantities.
- tool: a string, the tool used for crafting. Set to null if no tool is required.

`mine(object, tool)`
Mine the object with the tool: can only mine the object within reach, cannot mine object from a distance. If there are enough objects within reach, this action will mine as many as you specify. The obtained objects will be added to the inventory.
Augments:
- object: a string, the object to mine.
- tool: a string, the tool used for mining. Set to null if no tool is required.

`attack(object, tool)`
Attack the object with the tool: used to attack the object within reach. This action will keep track of and attack the object until it is killed.
Augments:
- object: a string, the object to attack.
- tool: a string, the tool used for mining. Set to null if no tool is required.

`equip(object)`
Equip the object from the inventory: used to equip equipment, including tools, weapons, and armor. The object must be in the inventory and belong to the items for equipping.
Augments:
- object: a string, the object to equip.

`digdown(object, tool)`
Dig down to the y-level with the tool: the only action you can take if you want to go underground for mining some ore.
Augments:
- object: an int, the y-level (absolute y coordinate) to dig to.
- tool: a string, the tool used for digging. Set to null if no tool is required.

`go_back_to_ground(tool)`
Go back to the ground from underground: the only action you can take for going back to the ground if you are underground.
Augments:
- tool: a string, the tool used for digging. Set to null if no tool is required.

`apply(object, tool)`
Apply the tool on the object: used for fetching water, milk, lava with the tool bucket, pooling water or lava to the object with the tool water bucket or lava bucket, shearing sheep with the tool shears, blocking attacks with the tool shield.
Augments:
- object: a string, the object to apply to.
- tool: a string, the tool used to apply.

2. You cannot define any new function. Note that the ”Generated structures” world creation option is turned off.

3. There is an inventory that stores all the objects I have. It is not an entity, but objects can be added to it or retrieved from it anytime at anywhere without specific actions. The mined or crafted objects will be added to this inventory, and the materials and tools to use are also from this inventory. Objects in the inventory can be directly used. Don’t write the code to obtain them. If you plan to use some object not in the inventory, you should first plan to obtain it. You can view the inventory as one of my states, and it is written in form of a dictionary whose keys are the name of the objects I have and the values are their quantities.

4. You will get the following information about my current state:
- inventory: a dict representing the inventory mentioned above, whose keys are the name of the objects and the values are their quantities
- environment: a string including my surrounding biome, the y-level of my current location, and whether I am on the ground or underground

Pay attention to this information. Choose the easiest way to achieve the goal conditioned on my current state. Do not provide options, always make the final decision.

5. You must describe your thoughts on the plan in natural language at the beginning. After that, you should write all the actions together. The response should follow the format:
{
"explanation": "explain why the last action failed, set to null for the first planning",
"thoughts": "Your thoughts on the plan in natural languag",
"action_list": [
{"name": "action name", "args": {"arg name": value}, "expectation": "describe the expected results of this action"},
{"name": "action name", "args": {"arg name": value}, "expectation": "describe the expected results of this action"},
{"name": "action name", "args": {"arg name": value}, "expectation": "describe the expected results of this action"}
]
}
The action_list can contain arbitrary number of actions. The args of each action should correspond to the type mentioned in the Arguments part. Remember to add "'dict"' at the beginning and the end of the dict. Ensure that you response can be parsed by Python json.loads

6. I will execute your code step by step and give you feedback. If some action fails, I will stop at that action and will not execute its following actions. The feedback will include error messages about the failed action. At that time, you should replan and write the new code just starting from that failed action.

A.3.2 User Query

USER:

My current state:
- inventory: {`inventory`}
- environment: {`environment`}

The goal is to {`goal`}.

Here is one plan to achieve similar goal for reference: {`reference plan`}.

Begin your plan. Remember to follow the response format.
or Action {`successful action`} succeeded, and {`feedback message`}. Continue your plan. Do not repeat successful action. Remember to follow the response format.
or Action {`failed action`} failed, because {`feedback message`}. Revise your plan from the failed action. Remember to follow the response format.

A.3.3 LLM Interface

Action Implementation. The observation of the action contains LiDAR rays with an interval of 5 degrees in the horizon and vertical direction for locating objects, and voxels with 10 unit radius only for navigation, inventory, life status, and agent location status (X-ray cheating is carefully avoided). RGB is not used in our implementation, although it provides more information than LiDAR rays. For example, the biome, and category of the dropping item can not be identified by LiDAR rays. Some objects may also be missed by LiDAR due to the sparseness of LiDAR rays. Different from Hafner et al. (2023) who set the breaking speed to 100, we did not change the game settings. The detailed implementation of each structured action is as follows:

- `equip`: The equip action calls the environment API to equip the required object. The action succeeds when the API returns success. The action fails when the object is not in inventory or the equip API returns failure.
- `explore`: The explore action traverses the world until the object is visible. This action regards the world as a chessboard, and each node on the chessboard is the center point of a 20×20 units

area. Two strategies are implemented depending on whether the agent is on the ground or not. When the agent is on the ground, the BFS explore will be adopted. When the agent is under the ground, mainly for exploring ore, the DFS explore will be adopted. In the DFS exploration, the agent will break the blocks to form a mine road with a width of 1 and a height of 2. The action succeeds when the object is visible. The action fails when the explore exceeds a preset steps of 10,000 but the required object is not found.

- `approach`: The approach action finds the nearest visible required object and walks towards the object. We adopt A^* algorithm for finding a path. The A^* algorithm can jump, translate, and fall in four directions of north, south, east and west. We also allow the agent to jump while placing a block under the agent for ascent. If the object is out of the voxel observation range, A^* algorithm is iteratively applied to find the location nearest to the object. The action succeeds when the ℓ^∞ norm distance between the object and agent is less than 2. The action fails when there is no required object visible or no path can be found to walk close to the object.
- `mine/attack`: The mine/attack action uses the keyboard attack API with the tools to attack the object. Only visible objects could be mined or attacked. The object of mine should be blocks, and the agent will continue mining the block until it is broken. The object of attack should be entities, and the agent will iteratively approach and attack the entity until it is killed. After the block is broken or the entity is killed, if there are items dropped by them, the agent will approach the items to collect them. The action succeeds when the block is broken or the entity is killed. The action fails when there is no visible object, no required tools is in inventory, or the visible object is out of attack range.
- `dig_down`: The dig_down action iteratively breaks the block underfoot with the tool until the required ylevel is reached. If the agent is on the ground, before digging down, the current location is stored for going up action. After the action succeeds, the state of the agent is set to underground. The action succeeds when the required ylevel is reached. The action fails when it exceeds the reset max steps 10,000 or no required tool is in inventory.
- `go_up`: The agent will first go back to the location stored by dig_down. Then, the go_up action puts dirt blocks underfoot to raise the agent. After the action is finished, the state of agent is set to on the ground. The action succeeds when the pre-stored location is reached. The action fails when the walk fails, exceeds the reset max steps 10,000 or there is no required tool in inventory.
- `build`: The build action places the required blocks according to a given blueprint from bottom to up. The action succeeds when all blocks have been placed. The action fails when there are no enough materials in inventory or it is invalid to place some blocks.
- `craft/smelt`: The action calls the environment API to craft/smelt the required object. The action succeeds when the required object is obtained. The actions fail when there are no enough materials in inventory or the agent is unable to place the crafting table/furnace or the API fails.
- `apply`: The apply action calls the keyboard use API, and applies the specific tool to the object, *e.g.*, applying the bucket on water to obtain water bucket. The action succeeds when the API returns success. The action fails when there is no visible object, no tool in inventory or the API fails.

Feedback Message. After the execution of each action, we will get feedback from the structured actions. The feedback will refresh the agent's state in Sec. A.3.2, including current inventory, biome, ylevel, and on/under the ground status. The feedback will also contain the success/fail message from these actions, as well as the inventory change during the action.

A.4 Memory

A.4.1 Learning Process

We maintain the text-based memory with a dictionary, whose keys are sub-goals and values are lists of successful action sequences for the corresponding sub-goals. The construction and update of the memory are through the following learning process:

- When encountering a new sub-goal that is not in the memory, the LLM planner creates plans without reference. Once the sub-goal is achieved, the entirely executed action sequence will be stored in the memory.

- When encountering a sub-goal with memory, the first action sequence in the recording list for this goal is retrieved as the reference plan, with which the LLM planner tries to achieve the goal. If it succeeds, the newly executed action sequence will be added to the last of the recording list.
- For each sub-goal, once the number of action sequences recorded in its list reaches N, we pop all the N sequences and use LLM to summarize them into a common plan solution suitable for various scenarios, which is then put first in the list. N is set to 5 in all our experiments.

To learn the memory for obtaining all items, starting from scratch each time would take a long time. In addition, it is necessary to avoid spending most of the time on learning simple tasks and not investing enough in learning difficult tasks. To improve learning efficiency, we suggest studying the sub-goals individually one by one. We first use our LLM Decomposer to generate sub-goal trees for all items, acquiring the set of all sub-goals involved. Then for each sub-goal, the LLM planner plays multiple times given its prerequisites including the required materials and tools. The learning process of the sub-goal is finished once we obtain $N = 5$ successful action sequences and summarize them into one common plan solution for reference.

A.4.2 Implementation of Memory Summarization

We also use `gpt-3.5-turbo` from OpenAI API for memory summarization but in a different dialogue. We use the following prompt to instruct the summarization with the role "SYSTEM". The slot {`action description`} is replaced with the same descriptions of interfaces of the structured actions as Sec. A.3.1. We list all the action sequences to be summarized in the query with the role "USER", which is fed into the LLM for response.

SYSTEM:

You serve as an assistant that helps me play the game Minecraft.

I am using a set of actions to achieve goals in the game Minecraft. I have recorded several action sequences successfully achieving a goal in a certain state. I will give you the goal, the state, and the sequences later. Please summarize the multiple action sequences into a single action sequence as a universal reference to achieve the goal given that certain state. Here are the instructions:

1. Each action sequence is a sequence of the following actions:

{`action description`}

2. The action sequences before and after summarization are always conditioned on the given state, i.e., the actions are taken in that certain state to achieve the goal. I will describe the state in the following form: State: - inventory: a dict whose keys are the name of the objects and the values are their quantities. This inventory stores all the objects I have. - environment: a dict including my surrounding biome and whether I am on the ground or underground.

3. The action sequence you summarize should be able to achieve the goal in general cases without specific modification. Every necessary action should be included, even though it does not appear in some sequences because I manually skipped it in some lucky cases. The actions redundant or irrelevant to the goal should be filtered out. The corner cases, such as success by luck and dealing with contingencies, should not be summarized into the final sequence.

4. You should describe your thoughts on summarization in natural language at the beginning. After that, give me the summarized action sequence as a list in JSON format. Your response should follow this form:

Thoughts: "Your thoughts and descriptions of your summarization"
Summarized action sequence:
[
{"name": "action name", "args": {"arg name": value}, "expectation": "describe the expected results of this action"},
{"name": "action name", "args": {"arg name": value}, "expectation": "describe the expected results of this action"},
{"name": "action name", "args": {"arg name": value}, "expectation": "describe the

```
expected results of this action"}
]
```

B Observation and Action Spaces

We list the observation and action spaces of different methods in Tab. 5. Prior RL-based agents take raw images as input and use low-level controls, while our agent accepts oracle inputs and uses structured actions. We only use voxel information of the blocks on the surface without X-ray cheating.

Table 5: **Observation and output spaces of different methods.**

Method	Perception Observation	Status Observation	Output Space
VPT	camera view RGB		keyboard/mouse (20 keys, mouse movements)
DreamerV3	camera view RGB	inventory life status	keyboard/mouse & GUI-free crafting (25 actions based on MineRL ObtainDiamond)
DEPS	camera view RGB block voxel (3 x 3 x 3)	yaw/pitch angle GPS location	keyboard/mouse & GUI-free crafting (42 actions discretized from MineDojo)
GITM (ours)	LiDAR rays (interval = 5") block voxel (radius = 10, without X-ray cheating)	inventory life status biome agent position	action APIs (9 APIs manually implemented on MineDojo)

C Results of All Items

We provide the success rate of all items in the entire Minecraft Overworld Technology Tree in Tab. 6.

Experiment Setting. Considering the large number of items, including those difficult to be obtained, we implemented an incremental testing strategy. This strategy is designed to keep the testing costs within a reasonable range, while also accounting for the rarity of certain items. We avoided a uniform increase in the number of tests across all items to accommodate the hardest-to-obtain ones, which would have resulted in prohibitive testing costs. Instead, we employed a incremental testing process.

For each item, we begin with 20 games. If the success count is less than or equal to 1, we increase to 50 games. If the success count remains less than or equal to 1, we further increase to 100, and eventually 200 games. This testing continues until the success count finally exceeds 1, or we complete 200 games. By following this efficient strategy, we ensure a cost-effective and reliable evaluation of each item, regardless of its availability. Moreover, because some items need long-term planning and crafting chain, we do not set restrictions on the time limit or query limit.

Exploring Biome. Biomes can be a key factor that strongly influences the success rate. Some items, like cactus, pumpkin, or melon, can only be found in specific biomes. The distribution of biomes highly limits the success rate of some items.

Table 6: **Success rate for all 262 items in the entire Minecraft Overworld Technology Tree.**

Item Name	Success Rate	Item Name	Success Rate	Item Name	Success Rate	Item Name	Success Rate
acacia boat	100	stonebrick	100	milk bucket	65	cactus	20
acacia door	100	trapdoor	100	coal block	65	activator rail	15
acacia fence	100	wooden axe	100	gravel	65	detector rail	15
acacia fence gate	100	wooden button	100	water bucket	60	diamond helmet	15
acacia stairs	100	wooden door	100	iron bars	60	slime ball	15
beef	100	wooden hoe	100	iron door	60	gold ingot	15
birch boat	100	wooden pickaxe	100	rail	60	gold nugget	15
birch door	100	wooden pressure plate	100	flower pot	60	gold ore	15
birch fence	100	wooden shovel	100	cauldron	60	golden shovel	15
birch fence gate	100	wooden slab	100	iron leggings	60	deadbush	15
birch stairs	100	wooden sword	100	flint	55	red mushroom block	15
boat	100	armor stand	100	arrow	55	golden hoe	15
bowl	100	rotten flesh	100	iron chestplate	55	golden sword	15
chest	100	stone slab	100	iron block	55	light weighted pressure plate	15
chicken	100	stone slab2	100	brick block	55	diamond leggings	15
cobblestone	100	red sandstone stairs	100	clay	55	pumpkin	15
cobblestone wall	100	sandstone stairs	100	hardened clay	55	pumpkin seeds	15
cooked beef	100	feather	100	red flower	50	brown mushroom block	15
cooked chicken	100	rabbit foot	100	yellow flower	50	mushroom stew	10
cooked mutton	100	item frame	95	egg	50	emerald	10
cooked porkchop	100	leather	95	hay block	45	lit pumpkin	10
crafting table	100	leather boots	95	flint and steel	45	golden axe	10
dark oak boat	100	leather helmet	85	hopper minecart	45	golden pickaxe	10
dark oak door	100	sapling	80	apple	45	golden boots	10
dark oak fence	100	tallgrass	80	beetroot	40	repeater	9
dark oak fence gate	100	wheat	80	beetroot seeds	40	carrot on a stick	9
dark oak stairs	100	wheat seeds	80	string	40	melon	8
dirt	100	iron ingot	80	diamond	40	melon seeds	8
double plant	100	iron nugget	80	diamond shovel	40	obsidian	7
fence	100	iron ore	80	jukebox	40	golden helmet	7
fence gate	100	iron shovel	80	bone	40	diamond chestplate	7
furnace	100	shield	80	bone meal	40	anvil	7
glass bottle	100	trapped chest	80	red mushroom	35	map	7
glass pane	100	tripwire hook	80	diamond hoe	35	writable book	6
jungle boat	100	grass	80	diamond sword	35	redstone block	6
jungle door	100	heavy weighted pressure plate	80	lava bucket	35	gunpowder	6
jungle fence	100	iron hoe	80	paper	35	bow	6
jungle fence gate	100	iron sword	80	reeds	35	golden carrot	5
jungle stairs	100	leaves	80	sugar	35	cake	4
ladder	100	painting	80	waterlily	35	sticky piston	4
lever	100	shears	80	baked potato	35	bone block	4
log	100	wool	80	potato	35	golden leggings	3
mutton	100	leather leggings	80	carrot	35	diamond block	3
oak stairs	100	coal	75	brown mushroom	35	clock	3
planks	100	torch	75	book	35	melon block	3
porkchop	100	snow	75	dropper	30	fermented spider eye	2
rabbit hide	100	snow layer	75	noteblock	30	pumpkin pie	2
red sandstone	100	snowball	75	redstone	30	golden rail	2
sandstone	100	bucket	75	redstone torch	30	fireworks	2
sign	100	iron axe	75	beetroot soup	30	lapis block	2
spruce boat	100	iron pickaxe	75	diamond axe	30	slime	2
spruce door	100	iron boots	75	diamond pickaxe	30	dispenser	1
spruce fence	100	iron trapdoor	75	bookshelf	25	golden chestplate	1
spruce fence gate	100	carpet	70	banner	25	gold block	1
spruce stairs	100	bed	70	diamond boots	25	speckled melon	1
stick	100	mossy cobblestone	70	fishing rod	25	lead	1
stone	100	vine	70	piston	25	poisonous potato	1
stone axe	100	brick	65	compass	20	rabbit stew	1
stone brick stairs	100	clay ball	65	brick stairs	20	emerald block	1
stone button	100	leather chestplate	65	spider eye	20	enchanting table	1
stone hoe	100	bread	65	lapis lazuli	20	golden apple	1
stone pickaxe	100	chest minecart	65	glass	20	enchanted book	0.5
stone pressure plate	100	furnace minecart	65	sand	20	tnt	0
stone shovel	100	hopper	65	ink sac	20	tnt minecart	0
stone stairs	100	iron helmet	65	cooked rabbit	20		
stone sword	100	minecart	65	rabbit	20		