A The `des-file` Format

A.1 Tutorial

As part of the release of MiniHack, we release an interactive tutorial jupyter notebook for the `des-file` format, which utilises MiniHack to visualise how the `des-file` affects the generated level.[9] Here we present an overview of the different kinds of `des-files`, how to add entities to levels, and the main sources of randomness that can be used to create a distribution of levels on which to train RL agents. An in-depth reference can also be found in [39].

A.2 Types of `des-files`

There are two types of levels that can be created using `des-file` format, namely MAZE-type and ROOM-type:

1. MAZE-type levels are composed of maps of the level (specified with the `MAP` command) which are drawn using ASCII characters (see Figure 3 lines 4-14), followed by descriptions of the contents of the level, described in detail below. In MAZE-type environments, the layout of the map is fixed, but random terrain can be created around (or within) that map using the `MAZEWALK` command, which creates a random maze from a given location and filling all available space of a certain terrain type.
2. ROOM-type levels are composed of descriptions of rooms (specified by the `ROOM` command), each of which can have its contents specified by the commands described below. Generally, the `RANDOM_CORRIDORS` command is then used to create random corridors between all the rooms so that they are accessible. On creation, the file specifies (or leaves random) the room's type, lighting and approximate location. It is also possible to create subrooms (using the `SUBROOM` command) which are rooms guaranteed to be within the outer room and are otherwise specified as normal rooms (but with a location relative to the outer room). Figure 15 illustrates an instance of the Oracle level specified by the ROOM-type `des-file` in Figure 14.

A.3 Adding Entities to `des-files`

As we have seen above, there are multiple ways to define the layout of a level using the `des-file` format. Once the layout is defined, it is useful to be able to add entities to the level. These could be monsters, objects, traps or other specific terrain features (such as sinks, fountains or altars). In general, the syntax for adding one of these objects is:

```
ENTITY: specification, location, extra-details
```

For example:

```
MONSTER: ('F',"lichen"), (1,1)
OBJECT: ('%',"apple"), (10,10)
TRAP: 'fire', (1,1)
# Sinks and Fountains have no specification
SINK: (1,1)
FOUNTAIN: (0,0)
```

As can be seen in Figure 14 lines 4-11, some objects (such as `statues`) have extra details that can be specified, such as the monster type (`montype`) of the statue. Note that many of the details here can instead be set to `random`, as shown for example in Figure 14, lines 29 and 33-36. In this case, the game engine chooses a suitable value for that argument randomly each time the level is generated. For monsters and objects, this randomness can be controlled by just specifying the class of the monster or object and letting the specific object or monster be chosen randomly. For example:

```
MONSTER: 'F', (1,1)
OBJECT: '%', (10,10)
```

[9] The tutorial can be found in MiniHack's documentation at `https://minihack.readthedocs.io`.

This code would choose a random monster from the Fungus class (https://nethackwiki.com/wiki/Fungus), and a random object from the Comestible class (https://nethackwiki.com/wiki/Comestible).

A.4 Sources of Randomness in des-file

We have seen how to create either very fixed (MAZE-type) or very random (ROOM-type) levels, and how to add entities with some degree of randomness. The des-file format has many other ways of adding randomness, which can be used to control the level generation process, including where to add terrain and in what way. Many of these methods are used in IF statements, which can be in one of two forms:

```
IF[50%] {
    MONSTER: 'F', (1,1)
} ELSE {
    # ELSE is not always necessary
    OBJECT: '%', (1,1)
}

IF[$variable_name < 15] {
    MONSTER: 'F', (1,1)
}
```

In the first form, a simple percentage is used for the random choice, whereas in the second, a variable (which could have been randomly determined earlier in the file) is used. A natural way to specify this variable is either in other conditional statements (perhaps you randomly add some number of monsters, and want to count the number of monsters you add such that if there are many monsters, you also add some extra weapons for the agent), or through dice notation. Dice notation is used to specify random expressions which resolve to integers (and hence can be used in any place an integer would be). They are of the form NdM, which means to roll N M-sided dice and sum the result. For example:

```
$roll = 2d6
IF[$roll < 7] {
    MONSTER: random, random
}
```

Dice rolls can also be used for accessing arrays, another feature of the des-file format. Arrays are initialised with one or more objects of the same type, and can be indexed with integers (starting at 0), for example:

```
# An array of monster classes
$mon_letters = { 'A', 'L', 'V', 'H' }
# An array of monster names from each monster class respectively
$mon_names = { "Archon", "arch-lich", "vampire lord", "minotaur" }
# The monster to choose
$mon_index = 1d4 - 1
MONSTER:($mon_letters[$mon_index],$mon_names[$mon_index]),(10,18)
```

Another way to perform random operations with arrays is using the SHUFFLE command. This command takes an array and randomly shuffles it. This would not work with the above example, as the monster name needs to match the monster class (i.e. we could not use ('A', "minotaur")). For example:

```
$object = object: { '[',')','*','%' }
SHUFFLE: $object
```

Now the $object array will be randomly shuffled. Often, something achievable with shuffling can also be achieved with dice-rolls, but it is simpler to use shuffled arrays rather than dice-rolls (for example, if you wanted to guarantee each of the elements of the array was used exactly once, but randomise the order, it is much easier to just shuffle the array and select them in order rather than try and generate exclusive dice rolls).

```
1  MAZE: "mylevel", ' '
2  GEOMETRY:center,center
3  MAP
4  ...........
5  ...........
6  ...........
7  ...........
8  ...........
9  ...........
10 ...........
11 ...........
12 ...........
13 ENDMAP
14 REGION:(0,0,11,9),lit,"ordinary"
15 REPLACE_TERRAIN:(0,0,11,9), '.', 'C', 33%
16 REPLACE_TERRAIN:(0,0,11,9), '.', 'T', 25%
17 TERRAIN:randline (0,9),(11,0), 5, '.'
18 TERRAIN:randline (0,0),(11,9), 5, '.'
19 $center = selection: fillrect (5,5,8,8)
20 $apple_location = rndcoord $center
21 OBJECT: ('%', "apple"), $apple_location
22
23 $monster = monster: { 'L','N','H','O','D','T' }
24 SHUFFLE: $monster
25 $place = { (10,8),(0,8),(10,0) }
26 SHUFFLE: $place
27 MONSTER: $monster[0], $place[0], hostile
28 STAIR:$place[2],down
29 BRANCH:(0,0,0,0),(1,1,1,1)
```

Figure 13: An example `des-file` based on the `HideNSeek` environment. Figure 22 presents several instances of environments generated using this `des-file`.

A.5 Random Terrain Placement

When creating a level, we may want to specify the structure or layout of the level (using a MAZE-type level), but then randomly create the terrain within the level, which will determine accessibility and observability for the agent and monsters in the level. As an example, consider the example `des-file` in Figure 13. In this level, we start with an empty 11x9 `MAP` (lines 3-13). We first replace 33% of the squares with clouds `C`, and then 25% with trees `T` (lines 15,16). To ensure that any corner is accessible from any other, we create two random-walk lines using `randline` from opposite corners and make all squares on those lines floor (`.`). To give the agent a helping hand, we choose a random square in the centre of the room with `rndcoord` (which picks a random coordinate from a selection of coordinates, see lines 19,20) and place an apple there.

Several other methods of randomly creating selections such as `filter` (randomly remove points from a selection) and `gradient` (create a selection based on a probability gradient across an area) are described in the NetHack wiki `des-file` format page [39].

```
LEVEL: "oracle"

ROOM: "ordinary" , lit, (3,3), (
    center,center), (11,9) {
  OBJECT:('`',"statue"),(0,0),
      montype:'C',1
  OBJECT:('`',"statue"),(0,8),
      montype:'C',1
  OBJECT:('`',"statue"),(10,0),
      montype:'C',1
  OBJECT:('`',"statue"),(10,8),
      montype:'C',1
  OBJECT:('`',"statue"),(5,1),
      montype:'C',1
  OBJECT:('`',"statue"),(5,7),
      montype:'C',1
  OBJECT:('`',"statue"),(2,4),
      montype:'C',1
  OBJECT:('`',"statue"),(8,4),
      montype:'C',1

  SUBROOM: "delphi" , lit , (4,3)
      , (3,3) {
    FOUNTAIN: (0, 1)
    FOUNTAIN: (1, 0)
    FOUNTAIN: (1, 2)
    FOUNTAIN: (2, 1)
    MONSTER: ('@', "Oracle"),
        (1,1)
    ROOMDOOR: false , nodoor ,
        random, random
  }

  MONSTER: random, random
  MONSTER: random, random

}
```

```
ROOM: "ordinary" , random,
    random, random, random {
  STAIR: random, up
  OBJECT: random,random
}

ROOM: "ordinary" , random,
    random, random, random {
  STAIR: random, down
  OBJECT: random, random
  TRAP: random, random
  MONSTER: random, random
  MONSTER: random, random
}

ROOM: "ordinary" , random,
    random, random, random {
  OBJECT: random, random
  OBJECT: random, random
  MONSTER: random, random
}

ROOM: "ordinary" , random,
    random, random, random {
  OBJECT: random, random
  TRAP: random, random
  MONSTER: random, random
}

ROOM: "ordinary" , random,
    random, random, random {
  OBJECT: random, random
  TRAP: random, random
  MONSTER: random, random
}

RANDOM_CORRIDORS
```

Figure 14: The ROOM-type des-file for the Oracle level in NetHack.

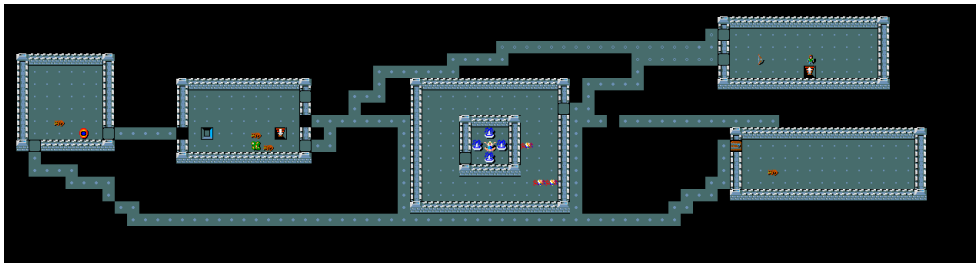

Figure 15: A screenshot of the `Oracle` level specified using the `des-file` in Figure 14.

B MiniHack

B.1 Observation Space

MiniHack, like NLE, has a dictionary-structured observation space. Most keys are inherited from NLE, while some are added in MiniHack. Note that using different observation keys can make environments significantly easier or harder (see Section 3.5 for discussion of how to ensure comparable experiments are performed).

- `glyphs` is a 21×79 matrix of glyphs (ids of entities) on the map. Each glyph represents an entirely unique entity, so these are integers between 0 and `MAX_GLYPH` (5991). In the standard terminal-based view of NetHack, these glyphs are represented by characters, with colours and other possible visual features.
- `chars` is a 21×79 matrix of the characters representing the map.
- `colors` is a 21×79 matrix of the colours of each of the characters on the map (some characters represent different objects or monsters depending on their colour).
- `specials` is a 21×79 matrix of special extra information about the view of that cell on the map, for example, if the foreground and background colour should be reversed.
- `blstats` ("Bottom Line Stats") is a representation of the status line at the bottom of the screen, containing information about the player character's position, health, attributes and other statuses. It comes in the form of a dimension 25 vector.
- `message` is the utf-8 encoding of the on-screen message displayed at the top of the screen. It's a 256-dimensional vector.
- `tty_chars` is the character representation of the entire screen, including the message and map, of size 24×80.
- `tty_colors` is the color representation of the entire screen, including the message and map, of size 24×80.
- `tty_cursor` is the location of the cursor on the screen, a 2-dimensional vector of (x,y) coordinates.
- `inv_glyphs` is a 55-dimensional vector representing the glyphs present in the current inventory view.
- `inv_letters` is a 55-dimensional vector representing the letters present in the current inventory view.
- `inv_oclasses` is a 55-dimensional vector representing the class of objects present in the current inventory view.
- `inv_strs` is a 55×80 matrix containing utf-8 encodings of textual descriptions of objects present in the current inventory view.

MiniHack adds the following observation keys:

- `screen_descriptions` is a $21 \times 79 \times 80$ tensor of utf-8 encodings of textual descriptions of each cell present in the map. NetHack provides these textual descriptions (which can be accessed by the user by using the describe action on a specific tile).
- `pixel`. We provide pixel-level observations based on the official NetHack tile-set. This is a representation of the current screen in image form, where each cell is represented by a 16x16x3 image, meaning the entire observation is so $336 \times 1264 \times 3$ (with 3 channels for RGB). Examples of this pixel observation can be seen in Figure 1 and Figure 6.
- Cropped observations. For `glyphs`, `chars`, `colors`, `specials`, `tty_chars`, `tty_colors`, `pixel`, and `screen_descriptions` a cropped observation centered the agent can be used by passing the observation name suffixed with `_crop` (e.g. `chars_crop`). This is a NxN matrix centered on the agent's current location containing the information normally present in the full view. The size of the crop can easily be configured using corresponding flags. Cropped observations can facilitate the learning, as the egocentric input makes representation learning easier.

B.2 Interface

There are two main MiniHack base classes to chose from, both derived from a common `MiniHack` base class.

`MiniHackNavigation` can be used to design mazes and navigation tasks that only require a small action space. All MiniHack navigation tasks we release, as well as MiniGrid and Boxoban examples, make use of the `MiniHackNavigation` interface. Here the pet is disabled by default. The in-game multiple-choice question prompts as well as menu navigation, are turned off by default.

`MiniHackSkill` provides a convenient mean for designing diverse skill acquisition tasks that require a large action space and more complex goals. All skill acquisition tasks in MiniHack use this

base class. The in-game multiple-choice question prompts is turned on by default, while the menu navigation is turned off. The player's pet and auto-pickup are disabled by default.

When designing environments, we particularly suggest using partially underspecified levels in order to use the built-in procedural content generator that changes the environment after every episode. For example, several aspects of the level described in Figure 3, such as the types and locations of monsters, objects, and traps, are not fully specified. The NetHack engine will make corresponding selections at random, making that specific feature of the environment procedurally generated. This enables a vast number of environment instances to be created. These could be instances the agent has never seen before, allowing for evaluation of test-time generalisation.

B.3 Level Generator

When creating a new MiniHack environment, a `des-file` must be provided. One way of providing this `des-file` is writing it entirely from scratch (for example files see Figure 14 and Figure 3, and for example environment creation see Figure 16a). However, this requires learning the `des-file` format and is more difficult to do programmatically, so as part of MiniHack we provide the `LevelGenerator` class which provides a convenient wrapper around writing a `des-file`. The `LevelGenerator` class can be used to create MAZE-type levels with specified heights and widths, and can then fill those levels with objects, monsters and terrain, and specify the start point of the level. Combined with the `RewardManager` which handles rewards (see Appendix B.4), this enables flexible creation of a wide variety of environments.

The level generator can start with either an empty maze (in which case only height and width are specified, see Figure 16b line 2) or with a pre-drawn map (see Figure 16c). After initialisation, the level generator can be used to add objects, traps, monsters and other terrain features. These can all be either completely or partially specified, in terms of class or location (see Appendix A.3 and Appendix A.4 for more information on how this works, and Figure 16b lines 6-10). Terrain can also be added programmatically at a later stage (Figure 16b lines 11-12). Once the level is complete, the `.get_des()` function returns the `des-file` which can then be passed to the environment creation function (Figure 16b lines 26-29).

B.4 Reward Manager

Along with creating the level layout, structure and content through the `LevelGenerator`, we also provide an interface to design custom reward functions. The default reward function of MiniHack is a sparse reward of +1 for reaching the staircase down (which terminates the episode), and 0 otherwise, with episodes terminating after a configurable number of time-steps. Using the `RewardManager`, one can control what events give the agent reward, whether those events can be repeated, and what combinations of events are sufficient or required to terminate the episode.

In Figure 16b lines 14-24 present an example of the reward manager usage. We first instantiate the reward manager and add events for eating an apple or wielding a dagger. The default reward is +1, and in general, all events are required to be completed for the episode to terminate. In lines 23-24, we add an event of standing on a sink which has a reward of -1 and is not required for terminating the episode.

While the basic reward manager supports many events by default, users may want to extend this interface to define their own events. This can be done easily by inheriting from the `Event` class and implementing the `check` and `reset` methods. Beyond that, custom reward functions can be added to the reward manager through `add_custom_reward_fn` method. These functions take the environment instance, the previous observation, action taken and current observation, and should return a float.

We also provide two ways to combine events in a more structured way. The `SequentialRewardManager` works similarly to the normal reward manager but requires the events to be completed in the sequence they were added, terminating the episode once the last event is complete. The `GroupedRewardManager` combines other reward managers, with termination conditions defined across the reward managers (rather than individual events). This allows complex conjunctions and disjunctions of groups of events to specify termination. For example, one could specify a reward

```python
# des_file is the path to the des-file
# or its content as a string
env = gym.make(
    "MiniHack-Navigation-Custom-v0",
    def_file=des_file)
```

(a) Creating a MiniHack navigation task via `des-file`.

```python
# Define a 10 by 10 grid area
lvl_gen = LevelGenerator(w=10, h=10)

# Populate it with different objects,
# a monster (goblin) and features (lava)
lvl_gen.add_object("apple", "%")
lvl_gen.add_object("dagger", ")")
lvl_gen.add_trap(name="teleport")
lvl_gen.add_sink()
lvl_gen.add_monster("goblin")
lvl_gen.fill_terrain("rect", "L",
                     0, 0, 9, 9)

# Define a reward manager
reward_manager = RewardManager()
# +1 reward and termination for eating
# an apple or wielding a dagger
reward_manager.add_eat_event("apple")
reward_manager.add_wield_event("dagger")
# -1 reward for standing on a sink
# but isn't required for terminating
# the episode
reward_manager.add_location_event("sink",
    reward=-1, terminal_required=False)

env = gym.make(
    "MiniHack-Skill-Custom-v0",
    des_file=lvl_gen.get_des(),
    reward_manager=reward_manager)
```

(b) Creating a skill task using the `LevelGenerator` and `RewardManager`.

```python
# Define the maze as a string
maze = """
--------------------
|.......|.|........|
|.-----.|.|-----|.|
|.|...|.|.|......|.|
|.|.|.|.|.|-----.|.|
|.|.|...|....|.|.|.|
|.|.--------.|.|.|.|
|.|..........|...|.|
|.|--------------|.|
|................|
--------------------
"""
# Set a start and goal positions
lvl_gen = LevelGenerator(map=maze)
lvl_gen.set_start_pos((9, 1))
lvl_gen.add_goal_pos((14, 5))
# Add a Minotaur at fixed position
lvl_gen.add_monster(name="minotaur",
    place=(19, 9))
# Add wand of death
lvl_gen.add_object("death", "/")

env = gym.make(
    "MiniHack-Skill-Custom-v0",
    des_file = lvl_gen.get_des())
```

(c) Creating a MiniHack skill task using the `LevelGenerator` with a pre-defined map layout.

Figure 16: Three ways to create MiniHack environments: using only a des-file, using the LevelGenerator and the RewardManager, and LevelGenerator with a pre-defined map layout.

function that terminates if a sequence of events (a,b,c) was completed, or all events {d,e,f} were completed in any order and the sequence (g,h) was completed.

B.5 Examples

Figure 16a presents how to create a MiniHack navigation task using only the `des-file`, as in Figure 3 or Figure 14.

Figure 16b shows how to create a simple skill acquisition task that challenges the agent to eat an apple and wield a dagger that is randomly placed in a 10x10 room surrounded by lava, alongside a goblin and a teleportation trap. Here, the `RewardManager` is used to specify the tasks that need to be completed.

Figure 16c shows how to create a labyrinth task. Here, the agent starts near the entrance of a maze and needs to reach its centre. A Minotaur is placed deep inside the maze, which is a powerful monster capable of instantly killing the agent in melee combat. There is a wand of death placed in a random location in the maze. The agent needs to pick it up, and upon seeing the Minotaur, zap it in the

direction of the monster. Once the Minotaur is killed, the agent needs to navigate itself towards the staircase (this is the default goal when `RewardManager` is not used).

C MiniHack tasks

This section presents detailed descriptions of existing MiniHack task and registered configurations. Tasks are grouped into similar tasks, within which several attributes are varied to make more difficult versions of the same task.

C.1 Navigation Tasks

Room. These tasks are set in a single square room, where the goal is to reach the staircase down (see Figure 17). There are multiple variants of this level. There are two sizes of the room (`5x5`, `15x15`). In the simplest variants, `Room-5x5` and `Room-15x15`), the start and goal position are fixed. In the `Room-Random-5x5` and `Room-Random-15x15` tasks, the start and goal position are randomised. The rest of the variants add additional complexity to the randomised version of the environment by introducing monsters (`Room-Monster-5x5` and `Room-Monster-15x15`), teleportation traps (`Room-Trap-5x5` and `Room-Trap-15x15`), darkness (`Room-Dark-5x5` and `Room-Dark-15x15`), or all three combined (`Room-Ultimate-5x5` and `Room-Ultimate-15x15`).[10]

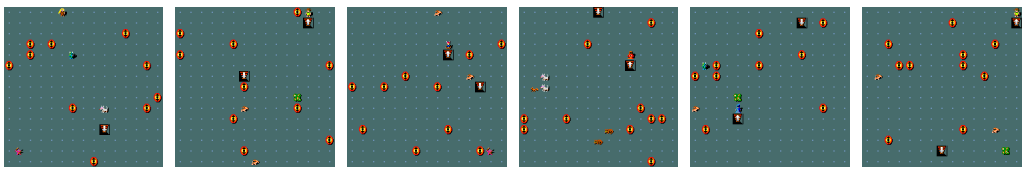

Figure 17: Various instances of the `Room-Ultimate-15x15` task.

Corridor. These tasks make use of the `RANDOM_CORRIDORS` command in the `des-file`. The objective is to reach the staircase down, which is in a random room (see Figure 18). The agent is also in a random room. The rooms have randomised positions and sizes. The corridors between the rooms are procedurally generated and are different for every episode. Different variants of this environment have different numbers of rooms, making the exploration challenge more difficult (`Corridor-R2`, `Corridor-R3`, and `Corridor-R5` environments are composed of 2, 3, and 5 rooms, respectively).

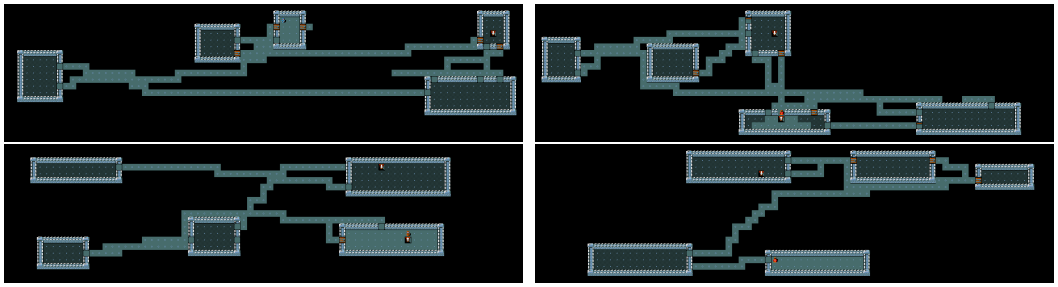

Figure 18: Four instances of the `Corridor-R5` task.

KeyRoom. These tasks require an agent to pick up a key, navigate to a door, and use the key to unlock the door, reaching the staircase down within the locked room. The action space is the standard movement actions plus the `PICKUP` and `APPLY` actions (see Figure 19). In the simplest variant of this task, (`KeyRoom-Fixed-S5`), the location of the key, door and staircase are fixed. In the rest of the variants, these locations randomised. The size the outer room is 5x5 for `KeyRoom-S5` and 15x15 for `KeyRoom-S15`. To increase the difficulty of the tasks, dark versions of the tasks are introduced

[10]The agent can attack monsters by moving towards them when located in an adjacent grid cell. Stepping on a lava tile instantly kills the agent. When the room is dark, the agent can only observe adjacent grid cells.

(KeyRoom-Dark-S5 and KeyRoom-Dark-S15), where the key cannot be seen if it is not in any of the agent's adjacent grid cells.

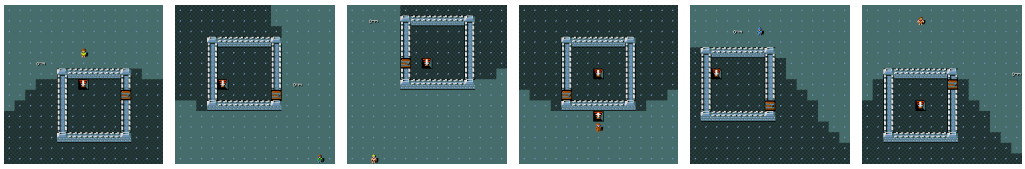

Figure 19: Various instances of the KeyRoom-S15 task.

MazeWalk. These navigation tasks make use of the MAZEWALK command in the des-file, which procedurally generates diverse mazes on the 9x9, 15x15 and 45x19 grids for MazeWalk-9x9, MazeWalk-15x15, and MazeWalk-45x19 environments, respectively (see Figure 20). In the mapped versions of these tasks (MazeWalk-Mapped-9x9, MazeWalk-Mapped-15x15, and MazeWalk-Mapped-45x19), the map of the maze and the goal's locations are visible to the agent.

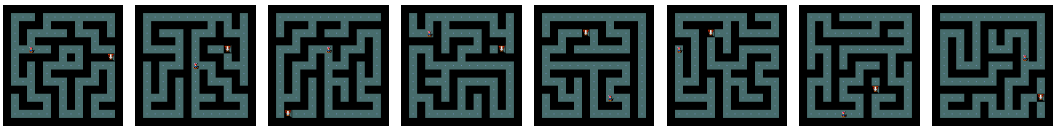

Figure 20: Various instances of the MazeWalk-15x15 task.

River. This group of tasks requires the agent to cross a river using boulders (see Figure 21). Boulders, when pushed into the water, create a dry land to walk on, allowing the agent to cross it. While the River-Narrow task can be solved by pushing one boulder into the water, other River require the agent to plan a sequence of at least two boulder pushes into the river next to each other. In the more challenging tasks of the group, the agent needs to additionally fight monsters (River-Monster), avoid pushing boulders into lava rather than water (River-Lava), or both (River-MonsterLava).

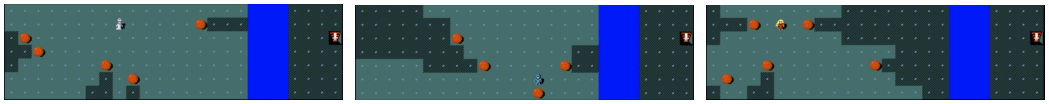

Figure 21: Three instances of the River task.

HideNSeek. In the HideNSeek task, the agent is spawned in a big room full of trees and clouds (see Figure 22). The trees and clouds block the line of sight of the player and a random monster (chosen to be more powerful than the agent). The agent, monsters and spells can pass through clouds unobstructed. The agent and monster cannot pass through trees. The goal is to make use of the environment features, avoid being seen by the monster and quickly run towards the goal. The layout of the map is procedurally generated, hence requires systematic generalisation. Alternative versions of this task additionally include lava tiles that need to be avoided (HideNSeek-Lava), have bigger size (HideNSeek-Big) or provide the locations of all environment features but not the powerful monster (HideNSeek-Mapped).

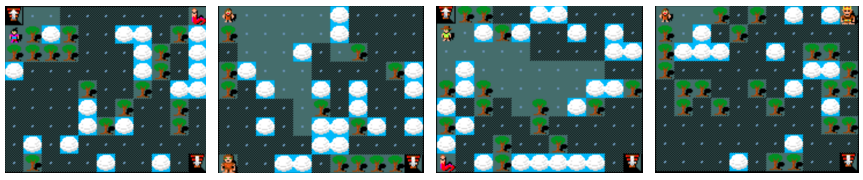

Figure 22: Four instances of the HideNSeek task.

CorridorBattle. The `CorridorBattle` task challenges the agent to make best use of the dungeon features to effectively defeat a horde of hostile monsters (see Figure 23). Here, if the agent lures the rats into the narrow corridor, it can defeat them one at a time. Fighting in rooms, on the other hand, would result in the agent simultaneously incurring damage from several directions and quick death. The task also is offered in dark mode (`CorridorBattle-Dark`), challenging the agent to remember the number of rats killed in order to plan subsequent actions.

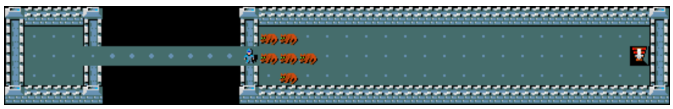

Figure 23: A screenshot of the `CorridorBattle` task.

Memento. This group of tasks test the agent's ability to use memory (within an episode) to pick the correct path. The agent is presented with a prompt (in the form of a sleeping monster of a specific type) and then navigates along a corridor (see Figure 24). At the end of the corridor, the agent reaches a fork and must choose a direction. One direction leads to a grid bug, which if killed terminates the episode with a +1 reward. All other directions lead to failure through an invisible trap that terminates the episode when activated. The correct path is determined by the cue seen at the beginning of the episode. We provide three versions of this environment: one with a short corridor before a fork with two paths to choose from (`Memento-Short-F2`), one with a long corridor with a two-path fork (`Memento-F2`), and one with a long corridor and a four-fork path (`Memento-F4`).

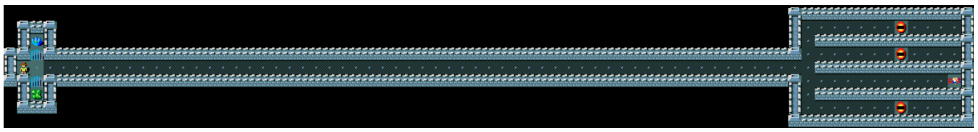

Figure 24: A screenshot of the `Memento-F4` task.

MazeExplore. These tasks test the agent's ability to perform deep exploration [42]. It's inspired by the Apple-Gold domain from [22], where a small reward can be achieved easily, but to learn the optimal policy deeper exploration is required. The agent must first explore a simple randomised maze to reach the staircase down, which they can take for +1 reward (see Figure 25). However, if they navigate through a further randomised maze, they reach a room with apples. Eating the apples gives a +0.5 reward, and once the apples are eaten the agent should then return to the staircase down. We provide an easy and a hard version of this task (`MazeExplore-Easy` and `MazeExplore-Hard`), with the harder version having a larger maze both before and after the staircase down. Variants can also be mapped (`MazeExplore-Easy-Mapped` and `MazeExplore-Hard-Mapped`), where the agent can observe the layout of the entire grid, making it easier to navigate the maze. Even in the mapped setting, the apples are not visible until the agent reaches the final room.

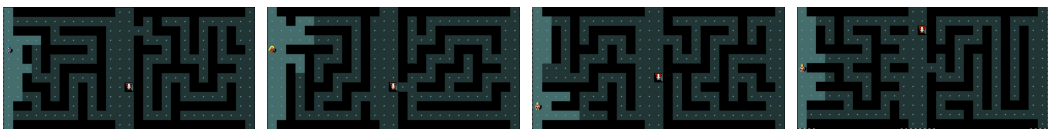

Figure 25: Four instances of the `MazeExplore-Hard` task. The apples are located near the right vertical wall (unobservable in the figure). The goal is located in the middle area of the grid.

The full list of navigation tasks in MiniHack is provided in Table 1.

C.2 Skill Acquisition Tasks

C.2.1 Skills

The nature of commands in NetHack requires the agent to perform a sequence of actions so that the initial action, which is meant for interaction with an object, has an effect. The exact sequence of

Table 1: Full list of MiniHack navigation tasks and corresponding capabilities for assessment.

Name	Capability
`Room-5x5-v0`	Basic Learning
`Room-15x15-v0`	Basic Learning
`Room-Random-5x5-v0`	Basic Learning
`Room-Random-15x15-v0`	Basic Learning
`Room-Dark-5x5-v0`	Basic Learning
`Room-Dark-15x15-v0`	Basic Learning
`Room-Monster-5x5-v0`	Basic Learning
`Room-Monster-15x15-v0`	Basic Learning
`Room-Trap-5x5-v0`	Basic Learning
`Room-Trap-15x15-v0`	Basic Learning
`Room-Ultimate-5x5-v0`	Basic Learning
`Room-Ultimate-15x15-v0`	Basic Learning
`Corridor-R2-v0`	Exploration
`Corridor-R3-v0`	Exploration
`Corridor-R5-v0`	Exploration
`KeyRoom-Fixed-S5-v0`	Exploration
`KeyRoom-S5-v0`	Exploration
`KeyRoom-Dark-S5-v0`	Exploration
`KeyRoom-S15-v0`	Exploration
`KeyRoom-Dark-S15-v0`	Exploration
`MazeWalk-9x9-v0`	Exploration & Memory
`MazeWalk-Mapped-9x9-v0`	Exploration & Memory
`MazeWalk-15x15-v0`	Exploration & Memory
`MazeWalk-Mapped-15x15-v0`	Exploration & Memory
`MazeWalk-45x19-v0`	Exploration & Memory
`MazeWalk-Mapped-45x19-v0`	Exploration & Memory
`River-Narrow-v0`	Planning
`River-v0`	Planning
`River-Monster-v0`	Planning
`River-Lava-v0`	Planning
`River-MonsterLava-v0`	Planning
`HideNSeek-v0`	Planning
`HideNSeek-Mapped-v0`	Planning
`HideNSeek-Lava-v0`	Planning
`HideNSeek-Big-v0`	Planning
`CorridorBattle-v0`	Planning & Memory
`CorridorBattle-Dark-v0`	Planning & Memory
`Memento-Short-F2-v0`	Memory
`Memento-F2-v0`	Memory
`Memento-F4-v0`	Memory
`MazeExplore-Easy-v0`	Deep Exploration
`MazeExplore-Hard-v0`	Deep Exploration
`MazeExplore-Easy-Mapped-v0`	Deep Exploration
`MazeExplore-Hard-Mapped-v0`	Deep Exploration

subsequent can be inferred by the in-game message bar prompts.[11] For example, when located in the same grid with an apple lying on the floor, choosing the `Eat` action will not be enough for the agent to eat it. In this case, the message bar will ask the following question: *"There is an apple here; eat it? [ynq] (n)*". Choosing the `Y` action at the next timestep will cause the initial `EAT` action to take effect, and the agent will eat the apple. Choosing the `N` action (or `MORE` action since `N` is the default choice) will decline the previous `EAT` action prompt. The rest of the actions will not progress the in-game timer and the agent will stay in the same state. We refer to this skill as `Confirmation`.

The `PickUp` skill requires picking up objects from the floor first and put in the inventory. The tasks with `InventorySelect` skill necessities selecting an object from the inventory using the corresponding key, for example, *"What do you want to wear? [fg or ?*]"* or *"What do you want to zap? [f or ?*]"*. The `Direction` skill requires choosing one of the moving directions for applying the selected action, e.g., kicking or zapping certain types of wands. In this case, *"In what direction?"* message will appear on the screen. The `Navigation` skill tests the agent's ability to solve various mazes and labyrinths using the moving commands.

C.2.2 Tasks

The full list of skill acquisition tasks, alongside the skills they require mastering, is provided in Table 2. The skill acquisition tasks are suitable testbeds for fields such as curriculum learning and transfer learning, either between different tasks within MiniHack or to the full game of NetHack.

Figure 26: Random instances of the `Eat-Distract`, `Wear-Distract` and `Pray-Distract` tasks.

Simple Tasks. The simplest skill acquisition tasks require discovering interaction between one object and the actions of the agent. These include: eating comestibles (`Eat`), praying on an altar (`Pray`), wearing armour (`Wear`), and kicking locked doors (`LockedDoors`). In the regular versions of these tasks, the starting location of the objects and the agent is randomised, whereas in the fixed versions of these tasks (`Eat-Fixed`, `Pray-Fixed`, `Wear-Fixed` and `LockedDoors-Fixed`) both are fixed. To add a slight complexity to the randomised version of these tasks, distractions in the form of a random object and a random monster are added to the third version of these tasks (`Eat-Distract`, `Pray-Distract` and `Wear-Distract`, see Figure 26). These tasks can be used as building blocks for more advanced skill acquisition tasks.

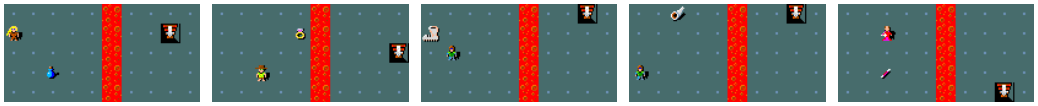

Figure 27: Five random instances of the `LavaCross` task, where the agent needs to cross the lava using (i) potion of levitation, (ii) ring of levitation, (iii) levitation boots, (iv) frost horn, or (v) wand of cold.

Lava Crossing. An example of a more advanced task involves crossing a river of lava. The agent can accomplish this by either levitating over it (via a potion of levitation or levitation boots) or freezing it (by zapping the wand of cold or playing the frost horn). In the simplest version of the task (`LavaCross-Levitate-Potion-Inv` and `LavaCross-Levitate-Ring-Inv`), the agent starts with one of the necessary objects in the inventory. Requiring the agent to pick up the corresponding object first makes the tasks more challenging (`LavaCross-Levitate-Potion-PickUp`

[11] Hence the messages are also used as part of observations in the skill acquisition tasks.

and LavaCross-Levitate-Ring-PickUp). The most difficult variants of this task group require the agent to cross the lava river using one of the appropriate objects randomly sampled and placed at a random location. In LavaCross-Levitate, one of the objects of levitation is placed on the map, while in the LavaCross task these include all of the objects for levitation as well as freezing (see Figure 27).

Figure 28: A screenshot of the WoD-Hard task.

Wand of Death. MiniHack is very convenient for making incremental changes to the difficulty of a task. To illustrate this, we provide a sequence of tasks that require mastering the usage of the wand of death [WoD, 38]. Zapping a WoD in any direction fires a death ray which instantly kills almost any monster it hits. In WoD-Easy environment, the agent starts with a WoD in its inventory and needs to zap it towards a sleeping monster. WoD-Medium requires the agent to pick it up, approach the sleeping monster, kill it, and go to the staircase. In WoD-Hard the WoD needs to be found first, only then the agent should enter the corridor with a monster (who is awake and hostile this time), kill it, and go to the staircase (see Figure 28). In the most difficult task of the sequence, the WoD-Pro, the agent starts inside a big labyrinth. It needs to find the WoD inside the maze and reach its centre, which is guarded by a deadly Minotaur.

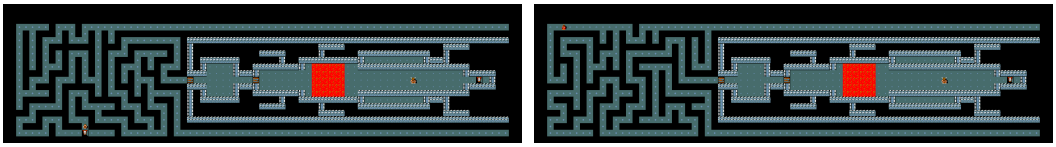

Figure 29: Two instances of the Quest-Hard task.

Quest. Quest tasks require the agent to navigate mazes, cross rivers and fight powerful monsters. Quest_Easy, the simplest environment in the group, challenges the agent to use objects in the inventory to cross the river of lava and fight a few relatively weak monsters. In Quest_Medium the agent engages with a horde of monsters instead and must lure them into the narrow corridors to survive. The Quest_Hard task, the most difficult environment of the group, requires the agent to solve complex, procedurally generated mazes, find objects for crossing the lava river and make use of the wand of death to kill a powerful monster (see Figure 29).

C.3 Ported Tasks

The full list of tasks ported to MiniHack from MiniGrid [12] and Boxoban [21] which we used in our experiments is provided in Table 3. Note that more tasks could have similarly been ported from MiniGrid. However, our goal is to showcase MiniHack's ability to port existing gridworld environments and easily enrich them, rather than porting all possible tasks.

D Experiment Details

Instructions on how to replicate experiments we present can be found in MiniHack's repository: https://github.com/facebookresearch/minihack. Below we provide details on individual components.

D.1 Agent and Environment Details

The agent architecture used throughout all experiments is identical to it in [33]. The observations used by the model include the 21×79 matrix of grid entity representations and a 21-dimensional vector containing agent statistics, such as its coordinates, health points, etc. Every of the 5991 possible

Table 2: Full list of MiniHack skill acquisition tasks.

Name	Skill
Eat-v0	Confirmation or PickUp+Inventory
Eat-Fixed-v0	Confirmation or PickUp+Inventory
Eat-Distract-v0	Confirmation or PickUp+Inventory
Pray-v0	Confirmation
Pray-Fixed-v0	Confirmation
Pray-Distract-v0	Confirmation
Wear-v0	PickUp+Inventory
Wear-Fixed-v0	PickUp+Inventory
Wear-Distract-v0	PickUp+Inventory
LockedDoor-v0	Direction
LockedDoor-Random-v0	Direction
LavaCross-Levitate-Ring-Inv-v0	Inventory
LavaCross-Levitate-Potion-Inv-v0	Inventory
LavaCross-Levitate-Ring-Pickup-v0	PickUp+Inventory
LavaCross-Levitate-Potion-PickUp-v0	PickUp+Inventory
LavaCross-Levitate-v0	PickUp+Inventory
LavaCross-v0	PickUp+Inventory
WoD-Easy	Inventory+Direction
WoD-Medium	PickUp+Inventory+Direction
WoD-Hard	PickUp+Inventory+Direction
WoD-Pro	Navigation+PickUp+Inventory+Direction
Quest-Easy-v0	Inventory
Quest-Medium-v0	Navigation+Inventory
Quest-Hard-v0	Navigation+PickUp+Inventory+Direction

Table 3: Tasks ported to MiniHack from other benchmarks.

Name	Capability
MultiRoom-N2-v0 [12]	Exploration
MultiRoom-N4-v0 [12]	Exploration
MultiRoom-N2-Monster-v0	Exploration
MultiRoom-N4-Monster-v0	Exploration
MultiRoom-N2-Locked-v0	Exploration
MultiRoom-N4-Locked-v0	Exploration
MultiRoom-N2-Lava-v0	Exploration
MultiRoom-N4-Lava-v0	Exploration
MultiRoom-N2-Extreme-v0	Exploration
MultiRoom-N4-Extreme-v0	Exploration
Boxoban-Unfiltered-v0 [21]	Planning
Boxoban-Medium-v0 [21]	Planning
Boxoban-Hard-v0 [21]	Planning

entities in NetHack (monsters, items, dungeon features, etc.) is mapped onto a k-dimensional vector representation as follows. First, we split the entity ids (glyphs) into one of twelve groups (categories of entities) and an id within each group. We construct a partition of the final vector which includes the following components as sub-vectors: group, subgroup_id, color, character, and special, each of which uses a separate embedding that is learned throughout the training. The relative length of each sub-vector is defined as follows: groups=1, subgroup_ids=3, colors=1, chars=2, and specials=1. That is, for a $k = 64$ dimensional embeddings, we make use of an embedding dimension of 24 for the id, 8 for group, 8 for color, 16 for character, and 8 for special. These settings were determined during a set of small-scale experiments.

Three dense representations are produced. First, all visible entity embeddings are passed to a CNN. Second, another CNN is applied to the 9×9 crop of entities surrounding the agent. Third, an MLP is used to encode the agent's statistic. These three vectors are concatenated and passed to another MLP which produces the final representation $\mathbf{o}_t$ of the observation. To obtain the action distribution, we feed the observations $\mathbf{o}_t$ to a recurrent layer comprised with an LSTM [26] cells, followed by an additional MLP.

For results on skill acquisition tasks, the in-game message, encoded using a character-level CNN [63], is also included as part of observation.

For all conducted experiments, a penalty of -0.001 is added to the reward function if the selected action of the agent does not increment the in-game timer of NetHack. For instance, when the agent attempts to move against a wall or navigates in-game menus, it will receive the -0.001 penalty.

D.2 TorchBeast Details

We employ an embedding dimension of 64 for entity representations. The size of the hidden dimension for the observation $\mathbf{o}_t$ and the output of the LSTM $\mathbf{h}_t$ is 256. We use a 5-layer CNN architecture (filter size 3×3, padding 1, stride 1) for encoding both the full screen of entities and the 9×9 agent-centred crop. The input channel of the first layer of the CNN is the embedding size of entities (64). The dimensions of all subsequent layers are equal to 16 for both input and output channels.

The characters within the in-game messages are encoded using an embedding of size 32 and passed to a 6-layer CNN architecture, each comprised of a 1D convolution of size 64 and ReLU nonlinearity. Maxpooling is applied after the first, second, and sixth layers. The convolutional layers are followed by an MLP.

We apply a gradient norm clipping of 40, but do not clip the rewards. We employ an RMSProp optimiser with a learning rate of $2 * 10^{-4}$ without momentum and with $\epsilon = 10^{-6}$.[12] The entropy cost is set to 10^{-4}. The γ discounting factor is set to 0.999.

The hyperparameters for Random Network Distillation [RND, 17] are the same as in [33] and mostly follow the author recommendations. The weights are initialised using an orthogonal distribution with gains of $\sqrt{2}$. A two-headed value function is used for intrinsic and extrinsic rewards. The discount factor for the intrinsic reward is set to 0.99. We treat both extrinsic and intrinsic rewards as episodic in nature. The intrinsic reward is normalised by dividing it by a running estimate of its standard deviation. Unlike the original implementation of RND, we do not use observation normalisation due to the symbolic nature of observations used in our experiments. The intrinsic reward coefficient is 0.1. Intrinsic rewards are not clipped.

For our RIDE [46] baselines, we normalise the intrinsic reward by the number of visits to a state. The intrinsic reward coefficient is 0.1. The forward and inverse dynamics models have hidden dimension of 128. The loss cost is 1 for the forward model and 0.1 for inverse model.

These settings were determined during a set of small-scale experiments.

The training on MiniHack's `Room-5x5` task for two million timesteps using our IMPALA baseline takes approximately 4:30 minutes (roughly 7667 steps per second). This estimate is measured using 2 NVIDIA Quadro GP100 GPUs (one for the learner and one for actors) and 20 Intel(R) Xeon(R) E5-2698 v4 @ 2.20GHz CPUs (used by 256 simultaneous actors) on an internal cluster. Our RND baseline completes the same number of timesteps in approximately 7:30 minutes (roughly 4092 steps per second) using the same computational resources.

[12] For results on skill acquisition tasks, we use a learning rate of $5 * 10^{-5}$.

D.3 Agent Architecture Comparison Details

Here we provide details on the architectures of models used in Figure 11. The *medium* model uses the exact architectures described in Appendix D.2, namely a 5-layer CNN, hidden dimension size 256, and entity embedding size 64. The *small* model uses a 3-layer CNN, hidden dimension size 64, and entity embedding size 16. The *large* model uses a 9-layer CNN, hidden dimension size 512, and entity embedding size 128. The rest of the hyperparameters are identical for all models and are described in Figure 11.

D.4 RLlib Details

We release examples of training several RL algorithms on MiniHack using RLlib [34]. RLlib is an open-source scalable reinforcement learning library built on top of Ray [37], that provides a unified framework for running experiments with many different algorithms. We use the same model as in the TorchBeast implementation described above, adjusted to not manage the time dimension for the models that use an LSTM (as RLlib handles the recurrent network logic separately). To enable future research on a variety of methods, we provide examples of training DQN [36], PPO [51] and A2C [35] on several simple MiniHack environments. Our DQN agent makes use of Double Q-Learning [55], duelling architecture [59] and prioritized experience replay (PER) [50]. We perform a limited sweep over hyperparameters, as we are not trying to achieve state-of-the-art results, just provide a starting point for future research.

In all experiments, we use 1 GPU for learning, and 10 CPUs for acting (with multiple instances of each environment per CPU), to speed up the wall-clock run-time of the experiments. These options can be configured easily in the code we release.

Results for these experiments can be seen in Figure 12. Hyperparameters for DQN, PPO, and A2C are detailed in Table 4, Table 5 and Table 6, respectively.

Table 4: RLlib DQN Hyperparameters

Name	value
learning rate	1e-6
replay buffer size	100000
PER β	0.4
n-step length	5
target network update frequency	50000
learning start steps	50000
PER annealing timesteps	100000

Table 5: RLlib PPO Hyperparameters

Name	value
learning rate	1e-5
batch size	128
SGD minibatch size	32
SGD iterations per epoch	2
rollout fragment length	128
entropy penalty coefficient	0.0001
value function loss coefficient	0.5
shared policy and value representation	True

Table 6: RLlib A2C Hyperparameters

Name	value
learning rate	1e-5
batch size	128
rollout fragment length	128
entropy penalty coefficient	0.001
value function loss coefficient	0.1

D.5 Unsupervised Environment Design

We base our UED experiments using PAIRED on MiniHack largely on those outlined in [15]. The hyperparameters used for training are provided in Table 7. Note that except where explicitly noted, all agents share the same training hyperparameters.

The adversary constructs new levels starting from an empty 5×5 grid. At each of the first 10 timesteps, the adversary chooses a position in which to place one of the following objects: {`walls`, `lava`, `monster`, `locked door`}. If a selected cell is already occupied, no additional object cell is placed. After placing the objects, the adversary then chooses the goal position followed by the agent's starting position. If a selected cell is already occupied, the position is randomly resampled from among the free cells.

At each time step, the adversary policy encodes the glyph observation using two convolution layers, each with kernel size 3×3, stride lengths of 1 and 2, and output channels, 16 and 32 respectively, followed by a ReLU activation over the flattened outputs. We embed the time step into a 10-dimensional space. The image embedding, time-step embedding, and the random noise vector are concatenated, and the combined representation is passed through an LSTM with a hidden dimension of 256, followed by two fully connected layers with a hidden dimension of 32 and ReLU activations to yield the action logits over the 169 possible cell choices.

We make use of the same architecture for the protagonist and antagonist policies, with the exceptions of using the agent-centred crop, rather than the full glyph observation, and producing policy logits over the MiniHack action space rather than over the set of possible cell positions.

Table 7: PAIRED hyperparameters

Name	value
γ	0.995
λ_{GAE}	0.95
PPO rollout length	256
PPO epochs	5
PPO minibatches per epoch	1
PPO clip range	0.2
PPO number of workers	32
Adam learning rate	1e-4
Adam ϵ	1e-5
PPO max gradient norm	0.5
PPO value clipping	yes
value loss coefficient	0.5
protagonist/antagonist entropy coefficient	0.0
adversary entropy coefficient	0.005

E Full Results

Figure 30, Figure 31, and Figure 32 present the results of baseline agents on all navigation, skill acquisition and ported MiniHack tasks, respectively.

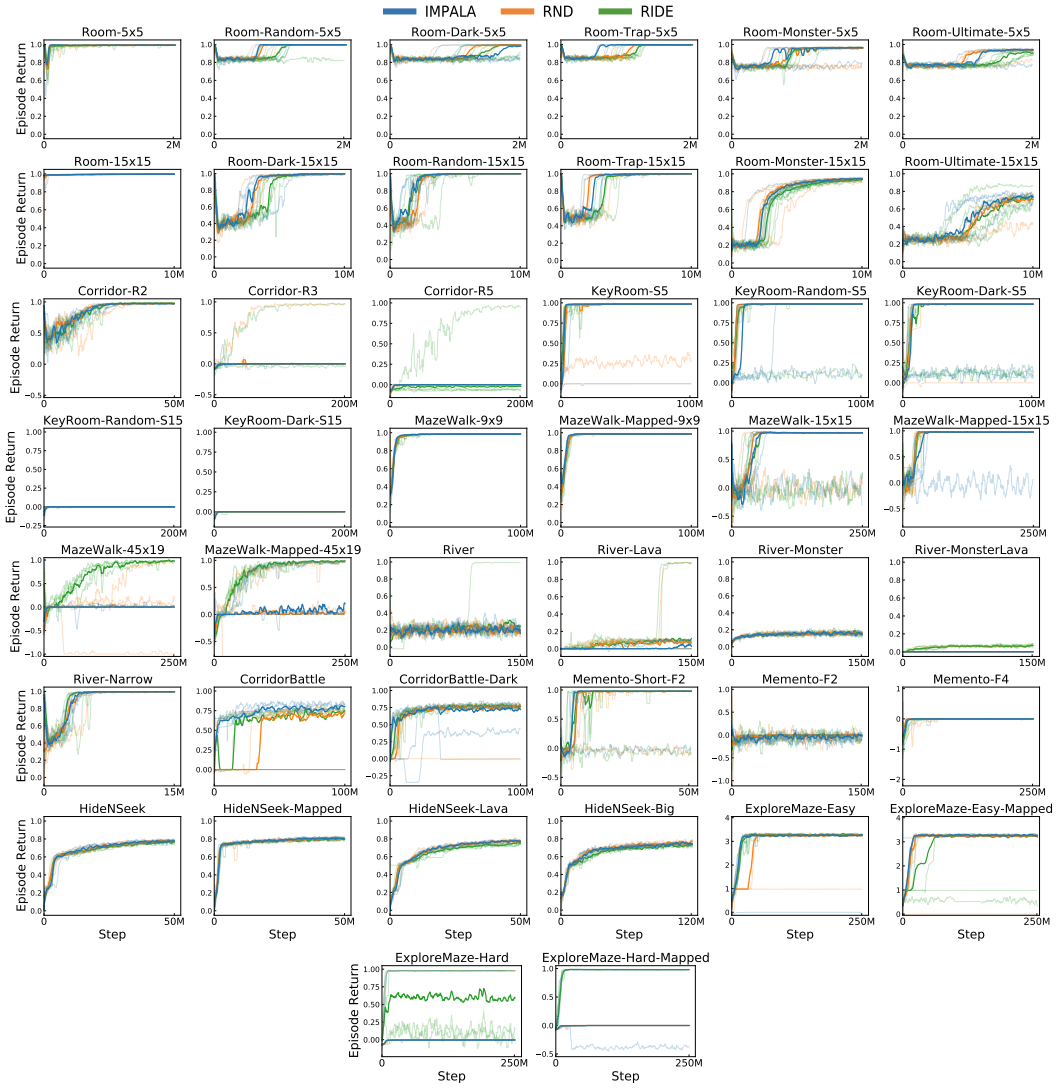

Figure 30: Mean episode returns on all MiniHack navigation tasks across five independent runs. The median of the runs is bolded.

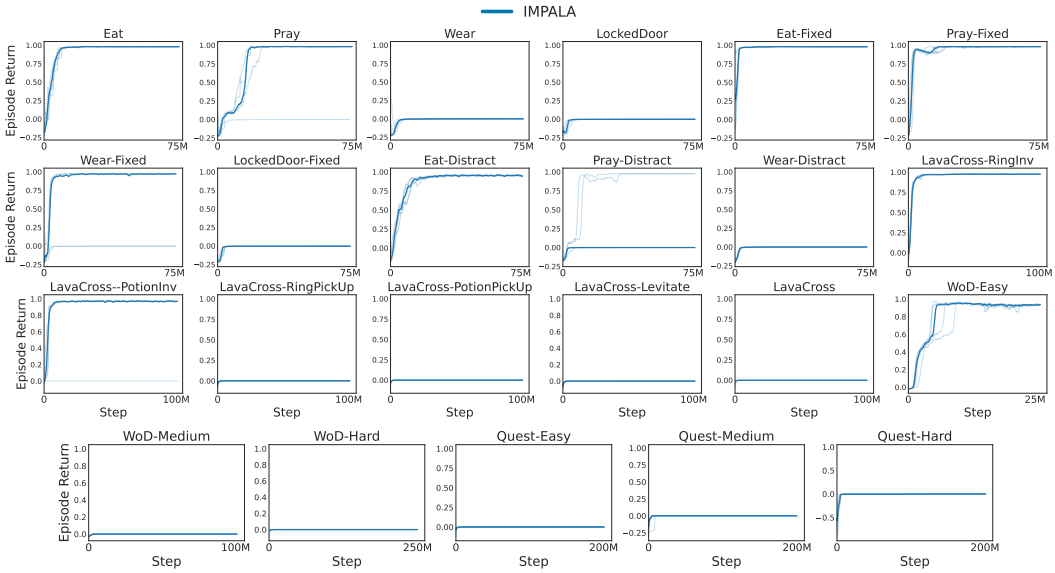

Figure 31: Mean episode returns on all MiniHack skill acquisition tasks across five independent runs. The median of the runs is bolded.

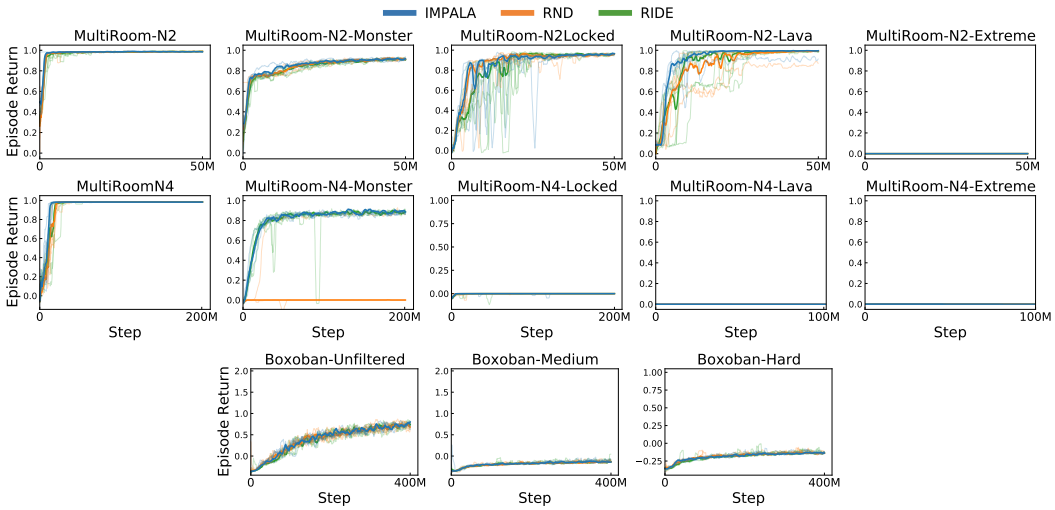

Figure 32: Mean episode returns on tasks ported to MiniHack from existing benchmarks. The median of the five runs is bolded.