MinePlanner: A Benchmark for Long-Horizon Planning in Large Minecraft Worlds

William Hill[*1], **Ireton Liu**[*1], **Anita De Mello Koch**[2], **Damion Harvey**[1], **Nishanth Kumar**[3], **George Konidaris**[2], **Steven James**[1]

[1]School of Computer Science and Applied Mathematics, University of the Witwatersrand
[2]Department of Computer Science, Brown University
[3]MIT Computer Science and Artificial Intelligence Laboratory

Abstract

We propose a new benchmark for planning tasks based on the Minecraft game. Our benchmark contains 45 tasks overall, but also provides support for creating both propositional and numeric instances of new Minecraft tasks automatically. We benchmark numeric and propositional planning systems on these tasks, with results demonstrating that state-of-the-art planners are currently incapable of dealing with many of the challenges advanced by our new benchmark, such as scaling to instances with thousands of objects. Based on these results, we identify areas of improvement for future planners. Our framework is made available at https://github.com/IretonLiu/mine-pddl/.

Introduction

A major challenge in AI is the construction of autonomous agents capable of solving extremely long-horizon tasks. While approaches such as reinforcement learning (RL) struggle with such tasks, especially with sparse feedback, task-level planners are well-suited to such problems. Additionally, these planners are typically domain-independent and so can be applied to a wide variety of problems, which is necessary if we desire generally intelligent agents.

However, these approaches require an abstract representation of a problem (typically using a structured language such as PDDL (McDermott et al. 1998)) as input. Furthermore, these representations are carefully crafted by a human designer to contain only the necessary information required to solve the task (Fishman et al. 2020). If we hope to scale these approaches to real-world tasks and develop truly autonomous agents, then planners must be capable of operating in domains that contain a large number of objects that may or may not be relevant to the task at hand.

While the issue of scaling to large domains is currently an area of active research (Illanes and McIlraith 2019), current planning domains continue to focus on simplified world models by simply increasing the number of objects present in standard benchmarks (Silver et al. 2021). This, however, fails to accurately reflect the difficulty a planner would face in a noisy real-world task. Additionally, recent work has demonstrated how PDDL representations can be directly learned from data (Asai and Fukunaga 2018; James, Rosman, and Konidaris 2020; Ahmetoglu et al. 2022; Silver et al. 2023). While these representations are often sound (Konidaris, Kaelbling, and Lozano-Pérez 2018), they typically contain many irrelevant symbols and action operators (James, Rosman, and Konidaris 2022); planners that are robust to this issue would further bridge the gap between learning and planning.

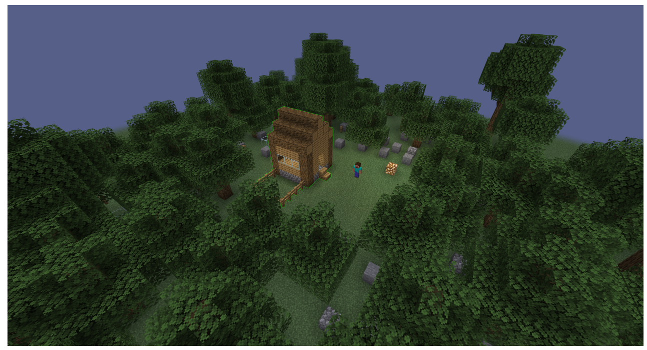


Figure 1: A classical planning problem in our benchmark, requiring the agent to collect the necessary blocks and build a log cabin (outlined in green). This task contains over 5000 objects that the agent must reason about, including many that make up the surrounding blocks and ground that are irrelevant to the goal.

Concurrently, the game of Minecraft has recently emerged as a promising testbed for RL research (Johnson et al. 2016), with its open-ended nature serving as a valuable proxy for the real world. As a domain, Minecraft has several desirable characteristics for planning research: (a) it supports a wide variety of tasks in the form of structures that can be assembled, all of which require long-term planning (see Figure 1 for an example); (b) the game is naturally populated by objects in the form of blocks and items, removing the need for a human designer to inflate the number of objects in the domain artificially; and (c) there is an inherent hierarchy in building large Minecraft structures (Beukman et al. 2023), which may be of interest to hierarchical planners (Höller et al. 2020).

In this paper, we present MinePlanner, a long-horizon planning benchmark in large Minecraft worlds. Our framework is capable of automatically generating tasks and verifying solutions in Minecraft, and supports both proposi-

*These authors contributed equally.

tional and numeric planners. We additionally provide a collection of 45 tasks which we used to benchmark representative propositional and numeric planners. These results show that there is significant work still left to be done for planning in large domains with many objects.

Related Work

Minecraft has proven to be a popular testbed for machine learning research (Johnson et al. 2016). One such platform is *MineRL* (Guss et al. 2019), which provides a set of Minecraft-related tasks for the agent to solve. More recently, Fan et al. (2022) incorporate background knowledge of the game in the form of online documentation, forums and videos of human gameplay to assist agents in learning to play the game. In both cases, they typically require an agent to act in a high-dimensional environment with partially observable pixel input. These benchmarks require agents to grapple with multiple problems simultaneously—high-dimensional function approximation, continuous control, partial observability and long-term planning. While a generally capable agent will ultimately be required to solve all of these problems, it also makes progress along any of these dimensions difficult to measure. Our proposed benchmark abstracts away these low-level intricacies, allowing researchers to focus on what is perhaps the most interesting aspect of Minecraft—the ability to create impressive structures, such as entire cities (Salge et al. 2020), through long-term planning.

Aluru et al. (2015) provide a framework for converting small, constrained Minecraft problems into object-oriented MDPs where A* search could be applied. However, only relevant objects were specified, and blocks that constituted the boundary walls and ground were not explicitly represented. By contrast, we represent open-world settings as generically as possible. Wichlacz, Torralba, and Hoffmann (2019) represent Minecraft construction tasks using PDDL and HTN formalisms; however, their representation abstracts much of the low-level complexity of Minecraft, most notably navigation, effectively treating the problem as a Blocks World. Finally, Roberts et al. (2017) automatically generate Minecraft representations using various extensions of PDDL to handle open worlds and partial observability, but constrain the observable range to make planning feasible.

One immediate challenge presented by Minecraft is the number of objects in the world. While previous work has identified the need to apply planners to domains with hundreds of objects, these domains are either scaled-up versions of classic problems such as *Blocksworld* (Silver et al. 2021) or are created by combining multiple existing planning problems to introduce irrelevant objects (Fishman et al. 2020). While these testbeds may be useful for developing better planning algorithms, the domains are disjoint from those considered by the RL community, squandering the opportunity for collaboration between the fields. Furthermore, these approaches may not capture the true complexity of the real world, which often contains objects and actions that may or may not be relevant to a given task.

Finally, while our benchmark hopes to spur research in planning, it can be combined with tools such as PDDLGym (Silver and Chitnis 2020) to act as a reinforcement learning environment. This would benefit RL researchers who wish to focus on the challenging long-term planning problem posed by Minecraft, while avoiding the complexity of continuous control in pixel space.

A Framework for Generating Minecraft Planning Tasks

We now present MinePlanner: a framework for generating Minecraft planning tasks that makes use of the APIs provided by MineDojo (Fan et al. 2022). At the highest level, we define a specification schema for tasks that are used to generate Minecraft worlds. We next extract objects and states (such as the agent's inventory) from the world and automatically generate a PDDL representation that can be used by planners. To provide support for multiple approaches, the framework can generate both numeric and propositional PDDL. The difference between the two is primarily how locations are represented, and we discuss this further in subsequent sections.

Our framework also supports the verification and visualisation of a plan—given the output of a planner, MinePlanner executes the proposed action in the game and verifies that the necessary predicates are achieved to solve the task. The frames collected during this process are saved and exported to video, which can then be used to promote research in a visually appealing manner. Finally, we provide a utility for extracting a list of objects, along with their coordinates, from saved Minecraft worlds[1] allowing users to easily specify new tasks without having to manually list the position of each object in the world.

Minecraft World Specification

We define a task as a set of blocks and items that are initially placed in the Minecraft world and the agent's inventory. For simplicity, we restrict the items to only those that can be placed in the world (e.g. wood blocks, but not pickaxes). We specify the goal of the task as a set of blocks that are to be placed in the world at some location, a set of items the agent must have in its inventory, the agent's location, or any combination of the three. An example of a task specification is shown in Listing 1, where an agent must place a log at location $(0, 4, 2)$ and additionally have at least one log remaining in its inventory to solve the task.

To produce a tractable representation of a Minecraft world, we make the following simplifications that vary slightly from the original game:

- Each task is created using a flat world with a single layer of grass blocks serving as the ground.
- There are no non-player characters, and items placed in the world do not despawn.[2]
- The agent does not require the necessary tools to break certain blocks. For example, the agent can break a tree block without an axe.

[1] Using the PyBlock library (github.com/alex4200/PyBlock).

[2] Both of these would violate the frame assumption (Pasula, Zettlemoyer, and Kaelbling 2004).

Listing 1: Example task specification in YAML.

```
name: "Example Problem"
blocks:
  - position:
      x: ’0’
      y: ’4’
      z: ’1’
    type: obsidian
items:
  - position:
      x: ’1’
      y: ’5’
      z: ’5’
    quantity: 1
    type: diamond
inventory:
  - type: log
    quantity: ’64’
  - type: obsidian
    quantity: ’64
goal:
  agent:
    - position:
        x: ’6’
        y: ’4’
        z: ’-5’
  blocks:
    - position:
        x: ’0’
        y: ’4’
        z: ’-2’
      type: log
  inventory:
    - type: log
      quantity: ’1’
```

- Broken blocks are immediately added to the agent’s inventory without being dropped on the ground as items.
- The agent is constrained by allowing it to move only one unit (block) in any cardinal direction at a given time.

State Representations

The types of each object, such as `agent`, `grass-block` or `flower`, are specified directly by Minecraft itself. To represent the state of the world, we must keep track of the location of all items and objects, as well as the agent’s inventory. Using PDDL 2.1 (Fox and Long 2003), this is relatively straightforward: the domain is defined by a predicate governing whether an object is present in the world (or in the agent’s inventory), or whether it has been destroyed, and several numeric fluents that keep track of each object’s x, y and z positions. Fluents are also used to track how many items are in the agent’s inventory, since the agent can collect multiple objects of the same type. There is one fluent for each type present in the world; for example, to track the number of flowers in the inventory, we would have the following: `(agent-num-flower ?ag - agent)`.

For planners that do not support numeric fluents, we represent positions and count using predicates only. This is achieved by defining “integer” objects (e.g., `position36`, `count0`) along with predicates that enforce relationships between these objects, such as sequentiality (`(are-seq ?x1 - int ?x2 - int)`) and whether an object is at a particular location (e.g., `(at-x ?l - locatable ?x - position)`). The inventory is represented by determining whether the count of a particular item matches some number.[3] The predicate equivalent to its numeric counterpart is `(agent-has-n-flower ?ag - agent ?n - count)`.

Operator Representations

We model two types of operators in our Minecraft worlds: *movement* and *interaction* actions. Movement involves the agent navigating one unit in a cardinal direction, and includes the ability to jump in a particular direction. As in the game, an agent’s movement is restricted by objects around it, and the preconditions for movement operators reflect this.

Another challenge in Minecraft is that there is no explicit action for picking up an item—an agent simply walks over an item to collect it. To avoid inconsistency between the PDDL representations and the game, we account for this by introducing two separate actions for every movement: a movement action that *cannot* be executed if the destination is occupied by an item and an action that combines a movement with a pickup. Additionally, the agent can only move to a destination above an existing block. The complete list of movement operators is as follows:

- `move_[direction]`: Moves the agent by one block in the specified direction. The agent cannot move to a location that is occupied by an item. An example of moving north without collecting an object is given by Listing 2.
- `move_and_pickup_[item]_[direction]`: Moves the agent by one block in the specified direction and collects an item at the resulting position.
- `jumpup_[direction]`: Moves the agent by one block in the specified direction and one block along the positive vertical axis. The agent cannot move into a position that is occupied by an item.
- `jumpdown_[direction]`: Moves the agent by one block in the specified direction and one block along the negative vertical axis. The agent cannot move into a position that is occupied by an item.
- `jumpup_and_pickup_[item]_[direction]`: Moves the agent by one block in the specified direction and one block along the positive vertical axis and collects an item at the resulting position.
- `jumpdown_and_pickup_[item]_[direction]`: Moves the agent by one block in the specified direction and one block along the negative vertical axis and collects an item at the resulting position.

The agent is also capable of manipulating blocks, and we support two such interaction operators:

- `place_[block]_[direction]`: Places a block from the agent’s inventory one block in front of the agent

[3] Minecraft enforces a maximum inventory count of 64 per object, which can be enumerated.

Listing 2: Example of a movement action using fluents

```
(:action move-north
 :parameters (?ag - agent)
 :precondition (and (agent-alive ?ag)
  (exists (?b - block) (and
  (block-present ?b) (= (x ?b) (x ?ag))
  (= (y ?b) (+ (y ?ag) -1)) (= (z ?b)
  (+ (z ?ag) -1)))) (and
  (not (exists (?b - block) (and
  (block-present ?b) (= (x ?b) (x ?ag))
  (or (= (y ?b) (+ (y ?ag) 1)) (= (y ?b) (y ?ag)))
  (= (z ?b) (+ (z ?ag) -1)))))
  (not (exists (?i - item) (and (item-present ?i) (= (x ?i) (x ?ag)) (= (y ?i) (y ?ag)) (= (z ?i)
  (+ (z ?ag) -1)))))))
 :effect (and (decrease (z ?ag) 1))
)
```

Listing 3: Example of an interaction action using predicates

```
(:action break-grass_block-north
 :parameters (?ag - agent ?b - grass_block-block ?x - position ?y - position ?y_up - position ?z - position ?z_front -
  position ?n_start - count ?n_end - count)
 :precondition (and (agent-alive ?ag)  (at-x ?ag ?x)  (at-y ?ag ?y) (at-z ?ag ?z)  (at-x ?b ?x) (at-y ?b ?y) (at-z ?b ?z_front) (are-seq ?z_front ?z) (are-seq ?y ?y_up)
  (block-present ?b) (not (exists (?i - item) (and (item-present ?i)
  (at-x ?i ?x) (at-y ?i ?y_up) (at-z ?i ?z_front)))) (are-seq ?n_start ?n_end) (agent-has-n-grass_block ?ag ?n_start)
  )
 :effect (and (not (block-present ?b))
  (not (at-x ?b ?x)) (not (at-y ?b ?y)) (not (at-z ?b ?z_front))
  (not (agent-has-n-grass_block ?ag ?n_start))
  (agent-has-n-grass_block ?ag ?n_end))
)
```

in the specified direction. The agent cannot place a block in a position that is occupied by another block or item and there must be a block below the target location.

- `break_[block]_[direction]`: Breaks the block one unit in front of the agent in the specified direction. The block is collected and added to the agent's inventory. The agent cannot break a block that has an item on top of it — in such a case, the agent will first have to pick the item up and then break the block. An example of breaking a grass block is given by Listing 3.

Goal Representations

Since Minecraft objects of the same type are interchangeable, we use existential preconditions when specifying a goal related to the location of a block. This allows us to specify that, for example, *any* wood block should be placed at a particular location, since all wood blocks are functionally identical. However, not all planners provide support for the `exist` keyword in the goal specification. To make our representation as accessible as possible, we introduce a "virtual" operator called `checkgoal` whose precondition is the actual condition for solving the task and whose effect sets a predicate `goal-achieved` to true. This is the only operator that can affect `goal-achieved`, which allows us to specify that the goal for all tasks is simply `goal-achieved`. For numeric planning, an example of a task whose goal is to place a `planks-block` at location $(0, 4, 2)$ is given by Listing 4, while Listing 5 shows the corresponding propositional equivalent.

Listing 4: Example of goal attainment using fluents

```
(:action checkgoal
 :parameters (?ag - agent)
 :precondition (and (agent-alive ?ag)
  (exists (?b - planks-block)
  (and (block-present ?b) (= (x ?b) 0)
  (= (y ?b) 4)(= (z ?b) 2))))
 :effect (and (goal-achieved ?ag))
)
```

Listing 5: Example of goal attainment using predicates

```
(:action checkgoal
 :parameters (?ag - agent)
 :precondition (and (agent-alive ?ag)
  (exists (?b - planks-block) (and
  (block-present ?b) (at-x ?b position0)
  (at-y ?b position4) (at-z ?b position2)))
  )
 :effect (and (goal-achieved ?ag))
)
```

Generating Solutions

Since we expect our tasks to be beyond the capabilities of current planners, we introduce a modification to Minecraft that allows human experts to play any of the defined tasks and generate a plan from their actions. The action trace of the player (consisting of player movements and interactions with blocks) is logged to a file that is then parsed into a format suitable for verification in MinePlanner. This subsystem serves two purposes: (a) it allows us to easily produce at least one satisficing plan, even in extremely complex tasks, and (b) it serves as a human benchmark against which the solutions produced by planners can be compared.

Benchmark Tasks in Minecraft

We create an initial suite of tasks, listed in Table 1, to serve as challenging problems for current planners. To create a fi-

Task	Variant	Observation Range	Inital Objects	Initial Predicates	Goal Pred.	Human Sol. Length	Description
move	Easy	$(13, 9, 13)$	0	762/851	1	5	Navigate to a specific location.
	Medium	$(21, 15, 21)$	12	1908/2273	1	11	
	Hard	$(71, 31, 71)$	1071	23706/29432	1	25	
pickup diamond	Easy	$(13, 9, 13)$	2	767/856	1	6	Navigate and pickup a single diamond in the world.
	Medium	$(21, 15, 21)$	8	1893/2254	1	7	
	Hard	$(71, 31, 71)$	1072	43870/54642	1	23	
gather wood	Easy	$(13, 9, 13)$	1	767/857	1	3	Navigate and pickup a single log in the world.
	Medium	$(21, 15, 21)$	6	1881/2240	1	4	
	Hard	$(71, 31, 71)$	1071	43859/54637	1	5	
place wood	Easy	$(13, 9, 13)$	1	767/856	2	3	Navigate to a specific location and place a log from inventory.
	Medium	$(21, 15, 21)$	13	1917/2284	2	11	
	Hard	$(71, 31, 71)$	1071	43863/54626	2	25	
pickup and place	Easy	$(13, 9, 13)$	1	767/857	1	16	First locate a plank in the world, then place it at a specific location.
	Medium	$(21, 15, 21)$	16	1926/2296	1	18	
	Hard	$(71, 31, 71)$	1071	43889/29432	1	42	
gather multi wood	Easy	$(13, 9, 13)$	3	775/867	1	7	Navigate and pickup a three logs in the world.
	Medium	$(21, 15, 21)$	18	1934/2306	1	20	
	Hard	$(71, 31, 71)$	1071	43865/54627	1	8	
climb	Easy	$(13, 9, 13)$	18	825/928	1	7	Place a block at an elevated y location by climbing a staircase.
	Medium	$(21, 15, 21)$	7	1892/2253	1	39	
	Hard	$(71, 31, 71)$	18	20387/25308	1	105	
cut tree	Easy	$(21, 31, 21)$	60	2113/2518	1	77	Cut down a tree by removing its wood.
	Medium	$(41, 31, 41)$	75	7144/8793	1	295	
	Hard	$(65, 31, 65)$	946	37080/46137	9	392	
build bridge	Easy	$(13, 9, 13)$	80	773/862	2	7	Build a wooden bridge over water.
	Medium	$(21, 15, 21)$	255	1924/2294	4	56	
	Hard	$(71, 31, 71)$	409	20350/25263	6	65	
build cross	Easy	$(13, 9, 13)$	5	788/882	5	45	Collect blocks to build a cross shape.
	Medium	$(21, 15, 21)$	10	1900/2264	5	84	
	Hard	$(71, 31, 71)$	1071	43854/54627	5	103	
build wall	Easy	$(13, 9, 13)$	9	806/904	9	117	Collect blocks to build a wall.
	Medium	$(21, 15, 21)$	16	1927/2298	9	99	
	Hard	$(71, 31, 71)$	1071	43869/54627	9	102	
build well	Easy	$(13, 9, 13)$	26	876/990	26	226	Collect blocks to build a well.
	Medium	$(21, 15, 21)$	36	2010/2400	26	443	
	Hard	$(71, 31, 71)$	1071	43870/54633	26	420	
build shape	Easy	$(13, 9, 13)$	1	772/861	5	31	Build a variety of shapes with items from inventory.
	Medium	$(21, 15, 21)$	12	1923/2288	9	74	
	Hard	$(71, 31, 71)$	1071	43871/54663	27	138	
collect and build shape	Easy	$(13, 9, 13)$	5	788/882	5	40	Collect blocks to build a variety of shapes.
	Medium	$(21, 15, 21)$	11	1910/2275	11	127	
	Hard	$(71, 31, 71)$	1071	43864/54647	27	292	
build cabin	Easy	$(21, 11, 21)$	0	1892/2246	116	481	Build a log cabin.
	Medium	$(41, 11, 41)$	116	7318/9008	116	904	
	Hard	$(65, 11, 65)$	5019	18201/22583	116	1268	

Table 1: A list of tasks provided by MinePlanner and their relevant statistics. Initial Objects does not include the ground (i.e. only objects explicitly specified in the YAML configuration are included). Initial Predicates is the number of predicates specified in the initial state of the problem file and is formatted as proposition/numerical. Goal Predicates is the number of goal conditions specified in the YAML configuration. Human Solution Length refers to the length of a plan constructed from a human expert's playthrough when solving the task.

nite representation of a (near infinite) Minecraft world, we consider only those blocks within some radius of the agent's initial location, termed the *observation range*.[4]

We define 15 types of tasks, where each task has three difficulties. Broadly, the easiest version of a task contains only those blocks necessary to solve the task and also has the smallest observation range. A medium difficulty task contains more blocks that typically exist in the world, but are not strictly relevant to the task at hand. Finally, the hardest version of each task takes place in a "realistic" Minecraft setting with a much larger observation range. The difference between difficulties is illustrated by Figure 2.

Experiments

We first benchmark Fast Downward (Helmert 2006), a propositional planner with the LAMA configuration, and ENHSP-20 (Scala et al. 2020), a numerical planner with A* search and the AIBR heuristic (Scala et al. 2016), on the tasks provided by MinePlanner. All experiments were conducted using Apptainer (Kurtzer, Sochat, and Bauer 2017) on a cluster of AMD Ryzen 9 7900X3D CPUs, using 128 virtual cores and 250GB of RAM per trial. We record planning time, including the amount of time spent preprocessing the PDDL file by each planner. For Fast Downward, this refers to the time taken to translate PDDL to SAS, while for ENHSP, this measures grounding. We set a timeout limit of two hours for each task, since an autonomous agent must ultimately be capable of planning in large environments within a reasonable timeframe. Results are reported in Table 2, with the means over five runs reported.[5]

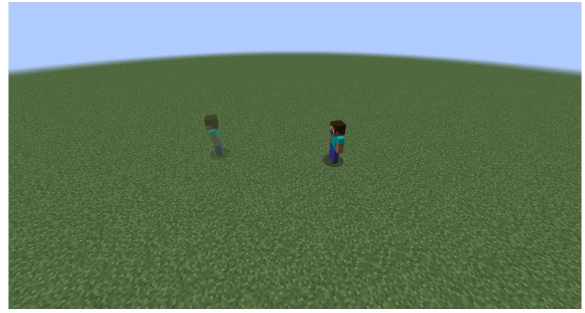
Domain-Independent Planning Results

The results indicate that the majority of tasks could not be solved by either planner. The translation step of Fast Downward was particularly problematic, with most of the tasks exhausting all memory before the file was translated to SAS. However, for those tasks where translation was successful, the subsequent search procedure was extremely fast (taking a few seconds at most). To investigate this behaviour, we conduct a further experiment using the `move` task. We begin with the Easy variant, and then incrementally scale the size of the world until Fast Downward fails to translate. These results, shown in Figure 3, how translation time scales exponentially as the size of the world increases. This is a worrying trend, despite the linear search time, as it indicates that Fast Downward does not scale well with the number of objects in the world.

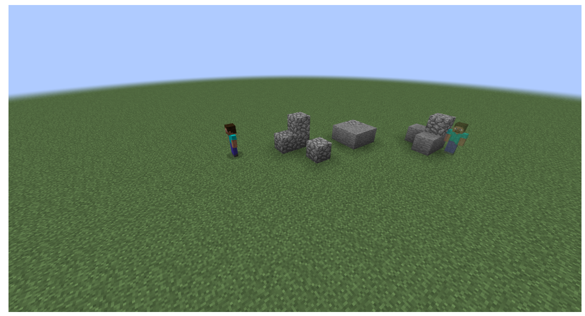
By contrast, ENHSP-20 is capable of grounding almost all of the problems, but fails to find a successful plan for most. For tasks that were successfully solved, planning is significantly slower than Fast Downward, illustrating the tradeoff between the two.

[4] The agent is centred in the *observation range*.

[5] For readability, we include only the means here, but report the full table along with standard deviations in the appendix.



(a) Easy variant



(b) Medium variant

(c) Hard variant

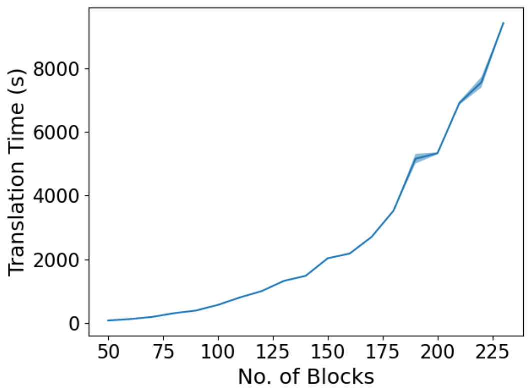
Figure 2: Three variants for the task of navigating to a particular location. (a) The easy task contains no irrelevant blocks, and so the world is empty. (b) The medium contains a few additional blocks which serve as obstacles and make navigation more challenging. (c) The hard task requires navigating within a small village consisting of hundreds of objects that are irrelevant for this particular task.

Automatically Removing Irrelevant Objects

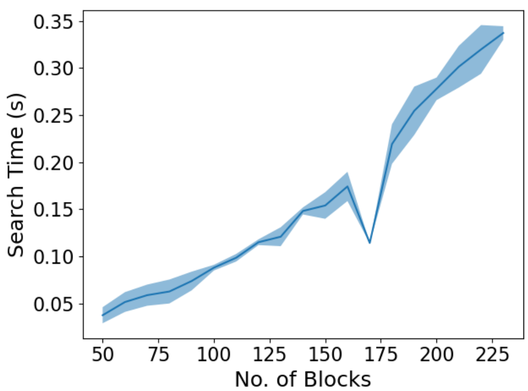
Given the issues caused by the number of objects in the world, it is natural to wonder whether many of these objects are irrelevant and can be pruned. We therefore apply *task scoping*, a recently introduced algorithm for preprocessing a PDDL file to remove provably irrelevant objects and operators, which has been shown to produce significant speedups in classic problems (Fishman et al. 2020). We apply this preprocessing step on the numeric domains that failed to plan in a reasonable amount of time. We select these tasks, since (a) task scoping requires the problem to first be preprocessed, so any tasks that could not complete the grounding step cannot

Task	Variant	FastDownward			ENHSP		
		Transl. (s)	Search (ms)	Total (s)	Ground (s)	Search (ms)	Total (s)
`move`	Easy	29.55	9.05	29.80	9.75	21.60	**25.19**
	Medium	—	—	—	49.05	> 7.2e6	> 7.2e6
	Hard	—	—	—	—	—	—
`pickup diamond`	Easy	242.85	172.88	**246.46**	9.28	7.15e6	7.19e6
	Medium	—	—	—	48.18	> 7.2e6	> 7.2e6
	Hard	—	—	—	—	—	—
`gather wood`	Easy	248.96	511.03	**251.35**	7.42	4 352.49	4 382.79
	Medium	—	—	—	47.32	> 7.2e6	> 7.2e6
	Hard	—	—	—	—	—	—
`place wood`	Easy	608.85	85420	**973.33**	7.53	> 7.2e6	> 7.2e6
	Medium	—	—	—	49.83	> 7.2e6	> 7.2e6
	Hard	—	—	—	—	—	—
`pickup and place`	Easy	597.39	525235	**1134.46**	7.36	> 7.2e6	> 7.2e6
	Medium	—	—	—	50.23	> 7.2e6	> 7.2e6
	Hard	—	—	—	—	—	—
`gather multi wood`	Easy	—	—	—	7.62	> 7.2e6	> 7.2e6
	Medium	—	—	—	50.19	> 7.2e6	> 7.2e6
	Hard	—	—	—	—	—	—
`climb`	Easy	—	—	—	8.70	> 7.2e6	> 7.2e6
	Medium	—	—	—	48.71	> 7.2e6	> 7.2e6
	Hard	—	—	—	—	—	—
`cut tree`	Easy	—	—	—	60.35	> 7.2e6	> 7.2e6
	Medium	—	—	—	—	—	—
	Hard	—	—	—	—	—	—
`build bridge`	Easy	—	—	—	13.38	> 7.2e6	> 7.2e6
	Medium	—	—	—	57.10	> 7.2e6	> 7.2e6
	Hard	—	—	—	—	—	—
`build cross`	Easy	—	—	—	8.31	> 7.2e6	> 7.2e6
	Medium	—	—	—	48.34	> 7.2e6	> 7.2e6
	Hard	—	—	—	—	—	—
`build wall`	Easy	—	—	—	8.48	> 7.2e6	> 7.2e6
	Medium	—	—	—	50.43	> 7.2e6	> 7.2e6
	Hard	—	—	—	—	—	—
`build well`	Easy	—	—	—	10.74	> 7.2e6	> 7.2e6
	Medium	—	—	—	54.40	> 7.2e6	> 7.2e6
	Hard	—	—	—	—	—	—
`build shape`	Easy	—	—	—	8.06	> 7.2e6	> 7.2e6
	Medium	—	—	—	51.70	> 7.2e6	> 7.2e6
	Hard	—	—	—	—	—	—
`collect and build shape`	Easy	—	—	—	8.19	> 7.2e6	> 7.2e6
	Medium	—	—	—	51.10	> 7.2e6	> 7.2e6
	Hard	—	—	—	—	—	—
`build cabin`	Easy	—	—	—	—	—	—
	Medium	—	—	—	—	—	—
	Hard	—	—	—	—	—	—

Table 2: The running times for Fast Downward and ENHSP-20 when run on the MinePlanner task suite. No result could be obtained for results marked as — because the planner failed to translate (in the case of Fast Downward) or ground (for ENHSP-20). Entries marked $> 7.2e6$ indicate that the planner timed out.



(a) Translation time for increasing world size.



(b) Planner search time for increasing world.size

Figure 3: The translation and planner search time is shown for the `move` task, starting from the size of the easy variant, and increasing until the FastDownward Planner can no longer translate the problem. The solid line and shaded areas represent the mean and standard deviation over five runs.

be scoped using this method, and (b) the removal of irrelevant objects should improve planner performance.

Unfortunately, our results indicate that the scoped environments have no effect on the final run time of the planner. Table 3 reports the average number of objects, actions and grounded actions removed across all tasks. In particular, task scoping fails to realise any advantages in the easy and medium task variants, since it considers the entire domain to be relevant and only removes grounded actions. We might expect to see gains for the hard variants, as they include many irrelevant objects. However, because the algorithm first requires that the problem be grounded, task scoping cannot be applied to these problems.

Lifted Planning

Since grounding tasks in MinePlanner has shown to be problematic, we also investigate whether planners that operate directly on the lifted representation could prove to be a solution. To this end, we attempt to run the Powerlifted planning system (Corrêa et al. 2020) in its recommended satisficing

Actions Removed	Objects Removed	Grounded Actions Removed
0 ± 0	0 ± 0	1009.25 ± 426

Table 3: The number of actions, objects and grounded actions removed by the task scoping algorithm. Results are averaged across all tasks in which the numeric planner failed to plan in a reasonable amount of time, with mean and standard deviation reported.

configuration on our problems.[6] This requires creating modified versions of all the operators in our domains to remove negations and existentials from the preconditions.

We found that Powerlifted quickly exhausts all 250GB of RAM during search on all of the easy variants of our tasks. As such, the planner is unable to find a solution to any of our tasks. This finding suggests that our tasks present significant challenges beyond existing benchmark problems for lifted planning (e.g. Lauer et al. (2021)), which Powerlifted is able to largely solve. Studying the source of the significant memory requirements for lifted planning in our problems could provide useful directions for future planning research.

Conclusion

We presented MinePlanner—a framework for generating Minecraft tasks in PDDL that can serve as challenging domains for classical planners. We also proposed a set of 45 initial tasks, varying in difficulty, and benchmarked two domain-independent planners on these domains. The results indicated that there is still a significant technical gap to overcome to solve these large problems, with no planner capable of solving any of the hard tasks.

One future direction is to leverage modern computing clusters to solve these challenging tasks, as has been done for previous "grand challenges" (Silver et al. 2016). However, at present, there are several issues that prevent the full utilisation of our hardware. While planners like ENHSP-20 could potentially benefit from parallelisation, especially due to their long search times, they are also not able to ground larger problems despite being provided with 250GB of RAM. Similarly, the tasks Fast Downward failed to complete were due to the high memory requirements needed during the translation phase. Modern planners will require more than parallelisation to scale effectively to object-dense environments.

If we wish to apply planning in realistic domains, we require planners that can operate within these object-dense environments in a reasonable amount of time, using a reasonable amount of memory. We hope that our benchmark will serve as a catalyst for developing new approaches to planning in complex domains, and form a bridge between the learning and planning communities.

[6]We also attempted to run QPlanner (Shaik and van de Pol 2021), but it was unable to parse and begin searching on even our easiest tasks.

References

Ahmetoglu, A.; Seker, M. Y.; Piater, J.; Oztop, E.; and Ugur, E. 2022. Deepsym: Deep symbol generation and rule learning for planning from unsupervised robot interaction. *Journal of Artificial Intelligence Research*, 75: 709–745.

Aluru, K. C.; Tellex, S.; Oberlin, J.; and MacGlashan, J. 2015. Minecraft as an experimental world for AI in robotics. In *AAAI Fall Symposium Series*.

Asai, M.; and Fukunaga, A. 2018. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In *AAAI Conference on Artificial Intelligence*, volume 32.

Beukman, M.; Fokam, M.; Kruger, M.; Axelrod, G.; Nasir, M.; Ingram, B.; Rosman, B.; and James, S. 2023. Hierarchically Composing Level Generators for the Creation of Complex Structures. *IEEE Transactions on Games*.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Frances, G. 2020. Lifted successor generation using query optimization techniques. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30.

Fan, L.; Wang, G.; Jiang, Y.; Mandlekar, A.; Yang, Y.; Zhu, H.; Tang, A.; Huang, D.-A.; Zhu, Y.; and Anandkumar, A. 2022. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *Advances in Neural Information Processing Systems*, 35: 18343–18362.

Fishman, M.; Kumar, N.; Allen, C.; Danas, N.; Littman, M.; Tellex, S.; and Konidaris, G. 2020. Task Scoping: Generating Task-Specific Abstractions for Planning in Open-Scope Models. *arXiv preprint arXiv:2010.08869*.

Fox, M.; and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of artificial intelligence research*, 20: 61–124.

Guss, W. H.; Houghton, B.; Topin, N.; Wang, P.; Codel, C.; Veloso, M.; and Salakhutdinov, R. 2019. MineRL: a large-scale dataset of Minecraft demonstrations. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, 2442–2448.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26: 191–246.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An extension to PDDL for expressing hierarchical planning problems. In *AAAI Conference on Artificial Intelligence*, volume 34, 9883–9891.

Illanes, L.; and McIlraith, S. 2019. Generalized planning via abstraction: arbitrary numbers of objects. In *AAAI Conference on Artificial Intelligence*, volume 33, 7610–7618.

James, S.; Rosman, B.; and Konidaris, G. 2020. Learning portable representations for high-level planning. In *International Conference on Machine Learning*, 4682–4691.

James, S.; Rosman, B.; and Konidaris, G. 2022. Autonomous learning of object-centric abstractions for high-level planning. In *International Conference on Learning Representations*.

Johnson, M.; Hofmann, K.; Hutton, T.; and Bignell, D. 2016. The Malmo Platform for Artificial Intelligence Experimentation. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, 4246–4247.

Konidaris, G.; Kaelbling, L. P.; and Lozano-Pérez, T. 2018. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research*, 61: 215–289.

Kurtzer, G. M.; Sochat, V.; and Bauer, M. W. 2017. Singularity: Scientific containers for mobility of compute. *PloS One*, 12(5).

Lauer, P.; Torralba, A.; Fišer, D.; Höller, D.; Wichlacz, J.; and Hoffmann, J. 2021. Polynomial-Time in PDDL input size: Making the delete relaxation feasible for lifted planning. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, 4119–4126.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language.

Pasula, H.; Zettlemoyer, L. S.; and Kaelbling, L. P. 2004. Learning Probabilistic Relational Planning Rules. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 73–82.

Roberts, M.; Piotrowski, W.; Bevan, P.; Aha, D.; Fox, M.; Long, D.; and Magazzeni, D. 2017. Automated planning with goal reasoning in Minecraft. In *Proceedings of ICAPS workshop on Integrated Execution of Planning and Acting*.

Salge, C.; Green, M. C.; Canaan, R.; Skwarski, F.; Fritsch, R.; Brightmoore, A.; Ye, S.; Cao, C.; and Togelius, J. 2020. The AI settlement generation challenge in Minecraft: First year report. *KI-Künstliche Intelligenz*, 34: 19–31.

Scala, E.; Haslum, P.; Thiebaux, S.; and Ramirez, M. 2016. Interval-based relaxation for general numeric planning. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, 655–663.

Scala, E.; Haslum, P.; Thiébaux, S.; and Ramirez, M. 2020. Subgoaling techniques for satisficing and optimal numeric planning. *Journal of Artificial Intelligence Research*, 68: 691–752.

Shaik, I.; and van de Pol, J. 2021. Classical Planning as QBF without Grounding. *arXiv preprint arXiv:2106.10138*.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587): 484–489.

Silver, T.; and Chitnis, R. 2020. PDDLGym: Gym Environments from PDDL Problems. In *International Conference on Automated Planning and Scheduling PRL Workshop*.

Silver, T.; Chitnis, R.; Curtis, A.; Tenenbaum, J. B.; Lozano-Pérez, T.; and Kaelbling, L. P. 2021. Planning with learned object importance in large problem instances using graph neural networks. In *AAAI Conference on Artificial Intelligence*, volume 35, 11962–11971.

Silver, T.; Chitnis, R.; Kumar, N.; McClinton, W.; Lozano-Pérez, T.; Kaelbling, L. P.; and Tenenbaum, J. 2023. Predicate Invention for Bilevel Planning. In *AAAI Conference on Artificial Intelligence*.

Wichlacz, J.; Torralba, A.; and Hoffmann, J. 2019. Construction-planning models in Minecraft. In *ICAPS Workshop on Hierarchical Planning*.

Appendix

A. Full Results

In Table 2, we benchmarked Fast Downward and ENHSP-20 on the tasks provided with MinePlanner. Table 4 below extends these results by reporting both the means and standard deviations over 5 runs. All experiments were conducted using Apptainer (Kurtzer, Sochat, and Bauer 2017) on a cluster of AMD Ryzen 9 7900X3D CPUs, using 128 virtual cores and 250GB of RAM.

Task	Variant	FastDownward			ENHSP		
		Transl. (s)	Search (ms)	Total (s)	Ground (s)	Search (ms)	Total (s)
`move`	Easy	29.55 ± 0.07	9.05 ± 3.69	29.80 ± 0.42	9.75 ± 0.02	21.60 ± 0.37	$\mathbf{25.19 \pm 0.48}$
	Medium	—	—	—	49.05 ± 0.48	$> 7.2e6$	$> 7.2e6$
	Hard	—	—	—	—	—	—
`pickup diamond`	Easy	242.85 ± 0.68	172.88 ± 1.62	$\mathbf{246.46 \pm 0.65}$	9.28 ± 0.09	$7.15e6 \pm 0.32e6$	$7.19e6 \pm 0.31e6$
	Medium	—	—	—	47.32 ± 0.29	$> 7.2e6$	$> 7.2e6$
	Hard	—	—	—	—	—	—
`gather wood`	Easy	248.96 ± 0.90	511.03 ± 3.0	$\mathbf{251.35 \pm 0.93}$	7.42 ± 0.42	$4\,352.49 \pm 76.25$	$4\,382.79 \pm 77.91$
	Medium	—	—	—	47.32 ± 0.36	$> 7.2e6$	$> 7.2e6$
	Hard	—	—	—	—	—	—
`place wood`	Easy	608.85 ± 2.62	85420 ± 28.4	$\mathbf{973.33 \pm 2.56}$	7.53 ± 0.07	$> 7.2e6$	$> 7.2e6$
	Medium	—	—	—	49.83 ± 0.23	$> 7.2e6$	$> 7.2e6$
	Hard	—	—	—	—	—	—
`pickup and place`	Easy	597.39 ± 1.88	$525\,235 \pm 392$	$\mathbf{1\,134.46 \pm 4.97}$	7.36 ± 0.27	$> 7.2e6$	$> 7.2e6$
	Medium	—	—	—	50.23 ± 0.68	$> 7.2e6$	$> 7.2e6$
	Hard	—	—	—	—	—	—
`gather multi wood`	Easy	—	—	—	7.62 ± 0.09	$> 7.2e6$	$> 7.2e6$
	Medium	—	—	—	50.91 ± 0.72	$> 7.2e6$	$> 7.2e6$
	Hard	—	—	—	—	—	—
`climb`	Easy	—	—	—	8.70 ± 0.07	$> 7.2e6$	$> 7.2e6$
	Medium	—	—	—	48.71 ± 0.14	$> 7.2e6$	$> 7.2e6$
	Hard	—	—	—	—	—	—
`cut tree`	Easy	—	—	—	60.35 ± 1.38	$> 7.2e6$	$> 7.2e6$
	Medium	—	—	—	—	—	—
	Hard	—	—	—	—	—	—
`build bridge`	Easy	—	—	—	13.38 ± 0.13	$> 7.2e6$	$> 7.2e6$
	Medium	—	—	—	57.10 ± 2.96	$> 7.2e6$	$> 7.2e6$
	Hard	—	—	—	—	—	—
`build cross`	Easy	—	—	—	8.31 ± 0.72	$> 7.2e6$	$> 7.2e6$
	Medium	—	—	—	48.34 ± 1.68	$> 7.2e6$	$> 7.2e6$
	Hard	—	—	—	—	—	—
`build wall`	Easy	—	—	—	8.48 ± 0.10	$> 7.2e6$	$> 7.2e6$
	Medium	—	—	—	50.43 ± 1.54	$> 7.2e6$	$> 7.2e6$
	Hard	—	—	—	—	—	—
`build well`	Easy	—	—	—	10.74 ± 0.08	$> 7.2e6$	$> 7.2e6$
	Medium	—	—	—	54.40 ± 1.21	$> 7.2e6$	$> 7.2e6$
	Hard	—	—	—	—	—	—
`build shape`	Easy	—	—	—	8.06 ± 0.83	$> 7.2e6$	$> 7.2e6$
	Medium	—	—	—	51.70 ± 0.72	$> 7.2e6$	$> 7.2e6$
	Hard	—	—	—	—	—	—
`collect and build shape`	Easy	—	—	—	8.19 ± 0.13	$> 7.2e6$	$> 7.2e6$
	Medium	—	—	—	51.10 ± 2.31	$> 7.2e6$	$> 7.2e6$
	Hard	—	—	—	—	—	—
`build cabin`	Easy	—	—	—	—	—	—
	Medium	—	—	—	—	—	—
	Hard	—	—	—	—	—	—

Table 4: The running times for Fast Downward and ENHSP-20 when run on the MinePlanner task suite. For results marked as —, no result could be obtained because the planner failed to translate (in the case of Fast Downward) or ground (for ENHSP-20). Entries marked $> 7.2e6$ indicate that the planner timed out. Means and standard deviations over 5 runs are reported.