Construction-Planning Models in Minecraft

Julia Wichlacz and **Álvaro Torralba** and **Jörg Hoffmann**
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
{wichlacz,torralba,hoffmann}@cs.uni-saarland.de

Abstract

Minecraft is a videogame that offers many interesting challenges for AI systems. In this paper, we focus on construction scenarios where an agent must build a complex structure made of individual blocks. As higher-level objects are formed of lower-level objects, the construction can naturally be modelled as a hierarchical task network. We model a house-construction scenario in classical and HTN planning and compare the advantages and disadvantages of both kinds of models.

Introduction

Minecraft is an open-world computer game, which poses interesting challenges for Artificial Intelligence (Aluru et al. 2015; Johnson et al. 2016), for example for the evaluation of reinforcement learning techniques (Tessler et al. 2017). Previous research on planning in Minecraft focused on models to control an agent in the Minecraft world. Some examples include learning planning models from a textual description of the actions available to the agent and their preconditions and effects (Branavan et al. 2012), or HTN models from observing players' actions (Nguyen et al. 2017). Roberts et al. (2017), on the other hand, focused on online goal-reasoning for an agent that has to navigate in the minecraft environment to collect resources and/or craft objects. They introduced several propositional, numeric (Fox and Long 2003) and hybrid PDDL+ planning models (Fox and Long 2006).

In contrast, we are interested in construction scenarios, where we generate instructions for making a given structure (e.g. a house) that is composed of atomic blocks. Our long-term goal is to design a natural-language system that is able to give instructions to a human user tasked with completing that construction. As a first step, in the present paper we consider planning methods coming up with what we call a *construction plan*, specifying the sequence of construction steps without taking into account the natural-language and dialogue parts of the problem.

For the purpose of construction planning, the Minecraft world can be understood as a Blocksworld domain with a 3D environment. Blocks can be placed at any position having a non-empty adjacent position. However, while obtaining a sequence of "put-block" actions can be sufficient for an AI agent, communicating the plan to a human user requires more structure in order to formulate higher-level instructions like *build-row*, or *build-wall*. The objects being constructed (e.g. rows, walls, or an entire house) are naturally organized in a hierarchy where high-level objects are composed of lower-level objects. Therefore, the task of constructing a high-level object naturally translates into a hierarchical planning network (HTN) (Sacerdoti 1974; Tate 1977; Wilkins 1988; Erol, Hendler, and Nau 1994).

We devise several models in both classical PDDL planning (Bylander 1994; McDermott et al. 1998) and hierarchical planning for a simple scenario where a house must be constructed. Our first baseline is a classical planning model that ignores the high-level objects and simply outputs a sequence of place-blocks actions. This is insufficient for our purposes since the resulting sequence of actions can hardly be described in natural language. However, it is a useful baseline to compare the other models. We also devise a second classical planning model, where the construction of high-level objects is encoded via auxiliary actions.

HTN planning, on the other hand, allows to model the object hierarchy in a straightforward way, where there is a task for building each type of high-level object. The task of constructing each high-level object can be decomposed into tasks that construct its individual parts. Unlike in classical planning, where the PDDL language is supported by most/all planners, HTN planners have their own input language. Therefore, we consider specific models for two individual HTN planners: the PANDA planning system (Bercher, Keen, and Biundo 2014; Bercher et al. 2017) and SHOP2 (Nau et al. 2003).

Scenario Design

We consider a simple scenario where our agent must construct a house in Minecraft. We model the Minecraft environment as a 3D grid, where each location is either empty or has a block of a number of types: wood, stone, or dirt.

Figure 1 shows the hierarchy of objects of our construction scenario. For the high-level structure the house consists of four stone walls, a stone roof, and a door. The walls and the roof are further decomposed into single rows that need to be built out of individual blocks. The door consists of two gaps, i.e., empty positions inside one of the walls.

As our focus is on the construction elements we abstract

low-level details away. For example, we avoid encoding the position of the agent and assume that all positions are always reachable. We also assume Minecraft's creative mode, where all block types are always available so we do not need to keep track of which blocks are there in the inventory.

This is a very simplistic model, where planning focuses simply on the construction actions (i.e. placing or removing blocks), of high-level structures. Nevertheless, it can still pose some challenges to modern planners, specially due to the huge size of the Minecraft environment.

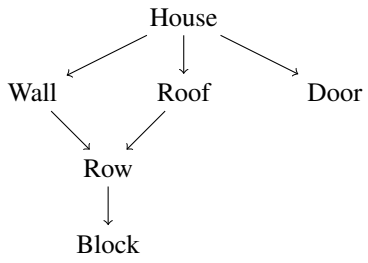

Figure 1: Object hierarchy of our construction scenario.

Classical Planning Model

Our first model is a classical planning model in the PDDL language that consists of only two actions: *put-block(?location, ?block-type)* and *remove-block(?location, ?block-type)* where there is a different location for each of the x-y-z coordinates in a 3D grid. The goal specifies what block-type should be in each location. As blocks cannot be placed in the air, the precondition of *put-block* requires one of the adjacent locations of *?location* to be non-empty. Other than that, blocks of any type can always be added or removed at any location. The goal is simply a set of *block_at* facts.

A limitation of this simple model is that it completely ignores the high-level structure of the objects being constructed. As there is no incentive to place blocks in certain order, a high-level explanation of the plan may be impossible. To address this, we introduce auxiliary actions that represent the construction of high-level objects. Figure 2 shows the auxiliary actions that represent building a wall. The attributes of the wall are specified in the initial state via attributes expressed by predicates *wall_dir*, *wall_length*, *wall_height*, *wall_type*, and *current_wall_loc*. In order to avoid the huge amount of combinations of walls that could be constructed of any dimensions and in any direction, the walls that are relevant for the construction at hand are specified in the initial state via these predicates. These three actions decompose the construction of a wall into several rows. Action *begin_wall* ensures that no other high-level object is being constructed at the moment and adds the fact *constructing_wall* to forbid the construction of any other wall (or roof) until the current wall has been finished.

Action *build_row_in_wall* ensures that a row of the given length will be built on the corresponding location and direction by adding predicates *(building_row)* and *(rest_row ?loc ?len ?dir ?t)*. Simultaneously, it updates the location for the rest of the wall to be built and decreases its height by one.

```
(:action begin_wall
  :parameters (?w - wall)
  :precondition (and (not (constructing_roof))
                     (not (constructing_wall))
  :effect (and (current_wall ?w) (constructing_wall)))

(:action build_row_in_wall
  :parameters (?w - wall ?loc ?locN - location
    ?len ?height ?heightN - number
    ?dir - direction ?t - blocktype)
  :precondition (and (current_wall ?w) (wall_dir ?w ?dir)
    (wall_length ?w ?len) (wall_height ?w ?height)
    (wall_type ?w ?t) (current_wall_loc ?w ?loc)
    (prev ?height ?heightN) (on_top ?loc ?locN)
    (not (building_row)))
  :effect (and (current_wall_loc ?w ?locN)
    (wall_height ?w ?heightN)
    (not (current_wall_loc ?w ?loc))
    (not (wall_height ?w ?height))
    (building_row) (rest_row ?loc ?len ?dir ?t)))

(:action finish_wall
  :parameters (?w - wall ?loc - location
    ?height - number ?dir - direction)
  :precondition (and (current_wall ?w) (is_zero ?height)
    (wall_height ?w ?height) (wall_dir ?w ?dir)
    (wall_initial ?w ?loc) (not(building_row)))
  :effect (and (wall_at ?w ?loc ?dir)
    (not (constructing_wall)) (not (current_wall ?w))))
```

Figure 2: Auxiliary PDDL actions to build a wall.

When the height is zero, the action *end_wall* becomes applicable, which finishes the construction of the wall.

In the goal we then use the predicates *wall_at* and *roof_at* that force the planner to use these constructions, instead of a set of *block_at* facts as we did in the simple model.

Hierarchical Planning Models

HTN models encode the construction of high-level objects in a straightforward way by defining tasks such as *build_house, build_wall* and *build_row*. These tasks will then be decomposed with methods until only primitive tasks will be left, in our case *place-block* and *remove-block*. We consider specific models for two individual HTN planners: the PANDA planning system (Bercher, Keen, and Biundo 2014; Bercher et al. 2017) and SHOP2 (Nau et al. 2003).

PANDA

PANDA uses an HTN formalism (Geier and Bercher 2011), which allows combining classical and HTN planning. The predicates describing the world itself, i.e. the relations between different locations remain the same as in the PDDL model, as do the *place-block* and *remove-block* primitive actions. On top of this, high-level objects are described as an HTN where each object corresponds to a task, without requiring to express their attributes with special predicates as we did in the PDDL model. Specifically, we defined tasks

```
(:method build_wall_1
  :parameters (?loc1 - location ?len ?hgt - numbers
    ?d - direction ?t - blocktype)
  :task (buildwall ?loc1 ?len ?hgt ?d ?t)
  :precondition (isone ?hgt)
  :subtasks (buildrow ?loc1 ?len ?d ?t))

(:method build_wall_2
  :parameters (?loc1 ?loc2 - location
    ?len ?hgt ?hgt2 - numbers
    ?d - direction ?t - blocktype)
  :task (buildwall ?loc1 ?len ?hgt ?d ?t)
  :precondition (and (not (isone ?hgt))
    (prev ?hgt ?hgt2)
    (on_top ?loc1 ?loc2))
  :ordered-subtasks (and
    (buildrow ?loc1 ?len ?d ?t)
    (buildwall ?loc2 ?len ?hgt2 ?d ?t)))
```

Figure 3: Methods for the build-wall task in the PANDA model.

that correspond to building a house, a wall, a roof, a row of blocks, and the door.

Figure 3 shows the methods used to decompose the task of building a wall. These methods work in a recursive fashion over the height of the wall. For walls with height one, the *build_wall_1* method is used to build them. For walls with larger height, the *build_wall_2* method decomposes the task of building them into building a row in the current location and building the rest of the wall (i.e., a wall of height-1) in the location above the previous one. These subtasks are ordered, so that walls are always built from bottom to top.

The methods for *buildrow* and *buildroof* work in the same fashion, while *buildhouse* only has one method decomposing the house into four walls, the roof, and the door. The task *builddoor* also has just one method stating which two blocks have to be removed to form a door. Choosing this way of modeling the door by first forcing the planner to place two blocks and later removing them again may seem inefficient, but for communication with a human user this may be preferable over indicating that these positions should remain empty in the first place.

SHOP2

The SHOP2 model follows a similar hierarchical task structure as the PANDA model, having methods for decomposing the house into walls, a wall into rows and rows into single blocks. Since one of the advantages of SHOP2 is that it can call arbitrary LISP functions, we can represent the locations using integers as coordinates and replace the predicates used in PANDA and PDDL to express their relations by simple arithmetic operations. This also allows us to compute the end point of rows of any given length in a given direction, which means we can construct the walls by alternating the direction of the rows. Based on this, we define two different recursive decompositions of walls as shown in Figure 4. In the first method we simply build the row starting in the current location, while in the second method we change the

```
(:method (build-wall-east ?x ?y ?z ?length ?height ?dir)
zero-height
  ((call = ?height 0))
  ()

east-one
  ((up ?z1 ?z) (up ?height ?h1) (call = ?dir 1))
  (:ordered
    (:task build-row ?x ?y ?z ?length ?dir)
    (:task build-wall-east ?x ?y ?z1 ?length ?h1 ?dir)
  )
)

(:method (build-wall-east ?x ?y ?z ?length ?height ?dir)
zero-height
  ((call = ?height 0))
  ()

 east-two
  ((up ?z1 ?z) (up ?height ?h1) (call = ?dir 1))
  (:ordered
    (:task build-row (call -(call + ?x ?length) 1) ?y ?z ?length 2)
    (:task build-wall-east ?x ?y ?z1 ?length ?h1 ?dir)
  )
)
```

Figure 4: SHOP2 methods to build a wall in east direction.

direction of the row we want to build and identify the position that would previously have been the end of the row by replacing the x-coordinate with $x + length - 1$. Since this computation is different for each direction, we need separate methods for them. Apart from this, the decomposition structure is the same as with PANDA, building the walls, roof, and rows incrementally using a recursive structure.

Experiments

To evaluate the performance of common planners on our models[1], we scale them with respect to two orthogonal parameters: the size of the construction, and the size of the cubic 3D world we are considering. We use different planners for each model. For the classical planning models we use the LAMA planner (Richter, Westphal, and Helmert 2011). The PANDA planning system implements several algorithms, including plan space POCL-based search methods (Bercher, Keen, and Biundo 2014; Bercher et al. 2017), SAT-based approaches (Behnke, Höller, and Biundo 2018), and forward heuristic search (Höller et al. 2018). We use a configuration using heuristic search with the FF heuristic, which works well on our models. For SHOP2, we use the depth-first search configuration (Nau et al. 2003). All experiments were run on an Intel i5 4200U processor with a time limit of 30 minutes and a memory limit of 2GB.

In our first experiment, we scale the size of the house starting with a $3 \times 3 \times 3$ house and increasing one parameter (length, width, and height) at a time ($4 \times 3 \times 3, 4 \times 4 \times$

[1]Benchmarks are publicly available at: `https://doi.org/10.5281/zenodo.3239243`

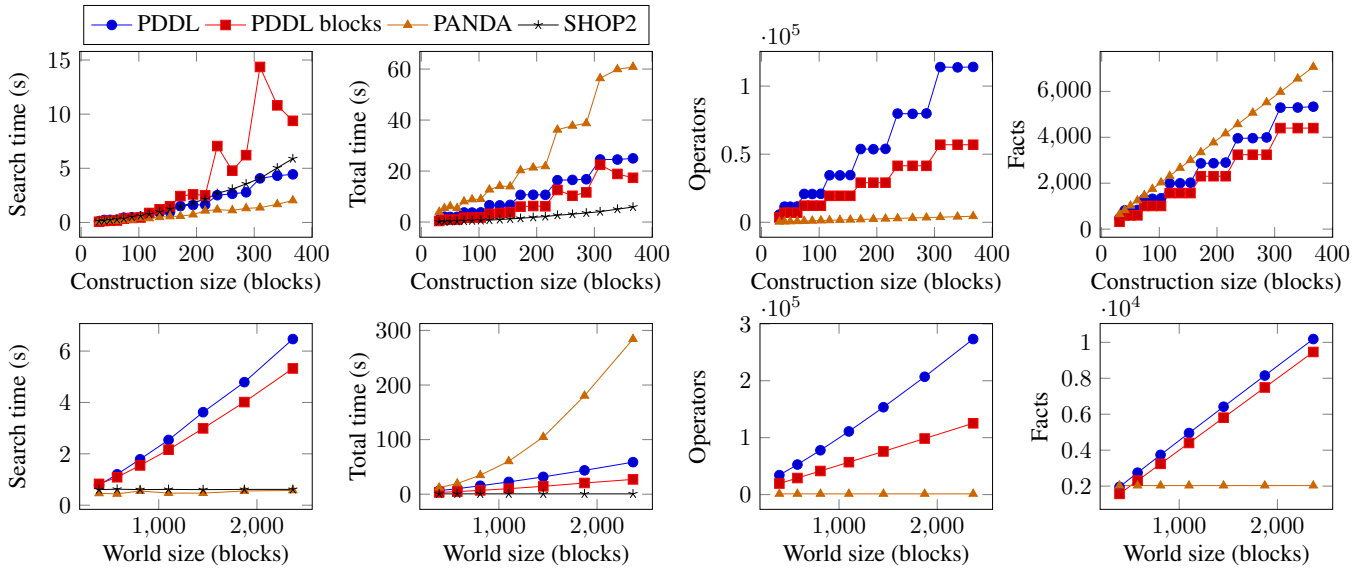

Figure 5: Search time, total time, number of operators, and facts of the grounded task to build a house with given number of blocks (above) or in a world with increasing size (below).

$3, \ldots, 9 \times 9 \times 9$.). The size of the 3D world is kept as small as possible to fit the house with some slack, so initially is set to $5 \times 5 \times 5$ and is increased by one unit in each direction every three steps, once we have scaled the house in all dimensions. The upper row of Figure 5 shows the search and total time of the planners on the different models. The construction size in the x-axis refers to the number of blocks that need to be placed in the construction. All planners scale well with respect to search time, solving problems of size up to $9 \times 9 \times 9$ in just a few seconds. The non-hierarchical PDDL planning model (PDDL blocks) that only uses the *place-block* and *remove-block* actions without any hierarchical information is the one with worst search performance. Moreover, it also results in typically longer plans that build many "support" structures to place a block in a wall without one of the adjacent blocks in the wall being there yet.

However, there is a huge gap between search and total time for the PANDA and PDDL models, mostly due to the overhead of the grounding phase. SHOP2 does not do any preprocessing or grounding so it is not impacted by this. For the PANDA and PDDL models, total time significantly increases every three problems, whenever the world size is increased. This suggests that, somewhat counterintuitively, the size of the world environment has a greater impact on these planners' performance than the size of the construction. In the PDDL based approaches, the number of operators and facts produced in the preprocessing shows a similar trend so the planner's performance seems directly influenced by the size of the grounded task. For PANDA, on the other hand, we observe a linear increase in the number of facts and only a comparatively small increase in the number of operators.

To test more precisely what is the impact of increasing the world size, we ran a second set of experiments where we kept the size of the house fixed at $5 \times 5 \times 5$ and just increased the size of the world. As shown in the bottom part of Figure 5 the performance of SHOP2 is not affected at all, since it does not require enumerating all possible locations. Search time for PANDA also stays mostly constant, but the overhead in the preprocessing phase dominates the total time. This contrasts with the number of operators and facts, which is not affected by the world size at all. The PDDL based models are also affected in terms of preprocessing time, due to a linear increase in the number of facts and operators with respect to world size, but to a lesser degree. However, search time increases linearly with respect to the world size due to the overhead caused in the heuristic evaluation.

Discussion

We have introduced several models of a construction scenario in the Minecraft game. Our experiments have shown that, even in the simplest construction scenario which is not too challenging from the point of view of the search, current planners may struggle when the size of the world increases. This is a serious limitation in the Minecraft domain, where worlds with millions of blocks are not unrealistic.

Lifted planners like SHOP2 perform well. However, it must be noted that they follow a very simple search strategy, which is very effective on our models where any method decomposition always leads to a valid solution. However, it may be less effective when other constraints must be met and/or optimizing quality is required. For example, if some blocks are removed from the ground by the user, then some additional blocks must be placed as auxiliary structure for the main construction. Arguably, this could be easily fixed by changing the model so that whenever a block cannot be placed in a target location, an auxiliary tower of blocks is built beneath the location. However, this increases the burden of writing new scenarios since suitable task decompositions (along with good criteria of when to select each decomposition) have to be designed for all possible situations.

This makes the SHOP2 model less robust to unexpected situations that were not anticipated by the domain modeler. PANDA, on the other hand, supports insertion of primitive actions (Geier and Bercher 2011), allowing the planner to consider placing additional blocks, e.g., to build supporting structures that do not correspond to any task in the HTN. This could help to increase the robustness of the planner in unexpected situations where auxiliary structures that have not been anticipated by the modeler are needed. However, this is currently only supported by the POCL-plan-based search component and considering all possibilities for task insertion significantly slows down the search and it runs out of memory in our scenarios. This may point out new avenues of research on more efficient ways to consider task insertion.

In related Minecraft applications, cognitive priming has been suggested as a possible solution to keep the size of the world considered by the planner at bay (Roberts and Hiatt 2017). In construction scenarios, however, large parts of the environment can be relevant so incremental grounding approaches may be needed to consider different parts of the scenario at different points in the construction plan.

Our models are still a simple prototype and they do not yet capture the whole complexity of the domain. We plan to extend them in different directions in order to capture how hard it is to describe actions or method decompositions in natural language. For example, while considering the position of the user is not strictly necessary, his visibility may be important because objects in his field of view are easier to describe in natural language. How to effectively model the field of vision is a challenging topic, which may lead to combinations with external solvers like in the planning modulo theories paradigm (Gregory et al. 2012).

Another interesting extension is to consider how easy it is to express the given action in natural language and for example by reducing action cost for placing blocks near objects that can be easily referred to. Such objects could be landmarks e.g. blocks of a different type ("put a stone block next to the blue block") or just the previously placed block (e.g., "Now, put another stone block on top of it").

References

Aluru, K. C.; Tellex, S.; Oberlin, J.; and MacGlashan, J. 2015. Minecraft as an experimental world for AI in robotics. 5–12. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT – totally-ordered hierarchical planning through SAT. In McIlraith, S., and Weinberger, K., eds., *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI'18)*, 6110–6118. AAAI Press.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An admissible HTN planning heuristic. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, 480–488. AAAI Press/IJCAI.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In Edelkamp, S., and Bartak, R., eds., *Proceedings of the 7th Annual Symposium on Combinatorial Search (SOCS'14)*. AAAI Press.

Branavan, S. R. K.; Kushman, N.; Lei, T.; and Barzilay, R. 2012. Learning high-level planning from text. 126–135. The Association for Computer Linguistics.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 69(1–2):165–204.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *Proceedings of the 12th National Conference of the American Association for Artificial Intelligence (AAAI'94)*, 1123–1129. Seattle, WA: MIT Press.

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.

Fox, M., and Long, D. 2006. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research* 27:235–297.

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In Walsh, T., ed., *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, 1955–1961. AAAI Press/IJCAI.

Gregory, P.; Long, D.; Fox, M.; and Beck, J. C. 2012. Planning modulo theories: Extending the planning paradigm. In Bonet, B.; McCluskey, L.; Silva, J. R.; and Williams, B., eds., *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018. A generic method to guide HTN progression search with classical heuristics. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS'18)*, 114–122. AAAI Press.

Johnson, M.; Hofmann, K.; Hutton, T.; and Bignell, D. 2016. The malmo platform for artificial intelligence experimentation. In Kambhampati, S., ed., *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI'16)*, 4246–4247. AAAI Press/IJCAI.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Comitee.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. Shop2: An htn planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Nguyen, C.; Reifsnyder, N.; Gopalakrishnan, S.; and Muñoz-Avila, H. 2017. Automated learning of hierarchical task networks for controlling minecraft agents. 226–231. IEEE.

Richter, S.; Westphal, M.; and Helmert, M. 2011. LAMA 2008 and 2011 (planner abstract). In *IPC 2011 planner abstracts*, 50–54.

Roberts, M., and Hiatt, L. M. 2017. Improving sequential decision making with cognitive priming. *Advances in Cognitive Systems*.

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*.

Sacerdoti, E. D. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5:115135.

Tate, A. 1977. Generating project networks. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI'77)*, 888–893. Cambridge, MA: William Kaufmann.

Tessler, C.; Givony, S.; Zahavy, T.; Mankowitz, D. J.; and Mannor, S. 2017. A deep hierarchical approach to lifelong learning in minecraft. In Singh, S., and Markovitch, S., eds., *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI'17)*, 1553–1561. AAAI Press.

Wilkins, D. E. 1988. *Practical Planning*. San Francisco, CA: Morgan Kaufmann.