Solving Tower of Hanoi in PDDL

Mihnea Tîrcă

December 21, 2022

Contents

1 Introduction

In this paper we will be solving the *Tower of Hanoi* problem using *Planning Domain Definition Language*. We will take a look at the game's rules (domain), as well as at our goal (problem). We will then transpose this problem into PDDL code. We will initially write the code for three disks (the original problem), and then we will try to generalize the problem by taking the number of disks as user input.

2 The game

The *Tower of Hanoi* is a mathematical game or puzzle consisting of three rods and a number of disks of various diameters, which can slide onto any rod. The puzzle begins with the disks stacked on one rod in order of decreasing size, the smallest at the top, thus approximating a conical shape. The objective of the puzzle is to move the entire stack to the last rod, obeying the following rules:

1. Only one disk may be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod
3. No disk may be placed on top of a disk that is smaller than it

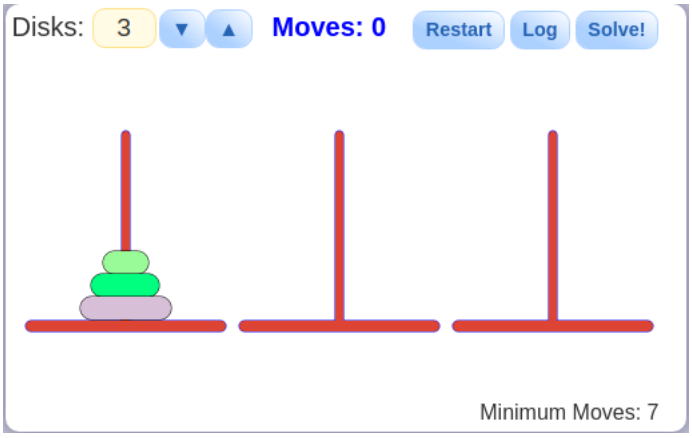

Figure 1: Tower of Hanoi starting position

With 3 disks, the puzzle can be solved in 7 moves. The minimal number of moves required to solve a *Tower of Hanoi* puzzle is

$$2^n - 1$$

, where n is the number of disks. We will not be taking a look at the math, rather solving the problem using PDDL.[1]

3 Coding the three disks problem in PDDL

3.1 The domain

To solve the problem using PDDL, we will first have to write the domain. We can think of the actions that we can take and then write the needed predicates accordingly. In theory, the only action that we can make is to move a disk. To move said disk to another tower, we would need to check if the disk is free (doesn't have another disk on top of it). So, we would need the predicate *(on-top ?d ?t)*, which checks if disk *d* is on top of tower *t*. We would obviously need the predicate *(smaller ?d1 ?d2)*, which checks if disk *d1* is smaller than disk *d2*, used when trying to move a disk on top of another. The predicates that I've found are needed are the following:

[1] To get a better understanding of the game, try playing it here

```
(:predicates
    (on-tower ?d ?t) ; disk d is on tower t (base of the tower)
    (on-disk ?d1 ?d2) ; disk d1 is on disk d2
    (disk ?d) ; d is a disk
    (tower ?t) ; t is a tower
    (empty ?t) ; tower t is empty
    (smaller ?d1 ?d2) ; disk d1 is smaller than disk d2
    (on-top ?d ?t) ; disk d is on top of tower t (free to move)
)
```

For the the action of moving a disk, we need to state that once we've moved disk A that used to sit on disk B, disk B is now free. But what if we move disk A that does not sit on a disk? We would need to write a separate action for this scenario. Thus, the scenarios are:

1. Moving a disk on top of a disk to a disk
2. Moving a disk on top of a disk to an empty tower
3. Moving a disk on an empty tower to a disk
4. Moving a disk on an empty tower to an empty tower

To code these actions, we need to check multiple preconditions:

1. The disk to move must always be free to move (on top of its tower).
2. If we move the disk to an empty tower, we must check that is is in fact empty.
3. If we move a disk A that sits on top of another disk B, disk B will be free after the move. Thus, we need to pass disk B as parameter and check in the preconditions if disk A sits on top of disk B. The effect will be that disk B is now free.
4. If we move a disk to another disk, we need to check that the disk to be moved is smaller.
5. If we move a disk from an empty tower, we need to check that it is at the base of its tower.

As for the effects, they are self-explanatory, given the preconditions.

3.1.1 Moving a disk on top of a disk to an empty tower

```
; move disk from disk to tower
; ?d is the disk to move
; ?df is the disk under disk ?d
; ?t1 is the tower ?d is on
; ?t2 is the tower to move ?d to
(:action move-d-to-t
    :parameters (?d ?df ?t1 ?t2)
    :precondition (and
        (disk ?d)
        (disk ?df)
        (tower ?t1)
        (tower ?t2)
        (empty ?t2)
        (on-disk ?d ?df)
        (on-top ?d ?t1)
        )
    :effect (and
        (on-top ?df ?t1)
        (not (empty ?t2))
        (not (on-disk ?d ?df))
```

```
        (on-tower ?d ?t2)
        (on-top ?d ?t2)
        (not (on-top ?d ?t1))
        )
)
```

3.1.2 Moving a disk on top of a disk to another disk

```
; move disk from disk to disk
; ?d is the disk to move
; ?df is the disk under disk ?d
; ?dt is the disk to move ?d to
; ?t1 is the tower ?d is on
; ?t2 is the tower ?dt is on
(:action move-d-to-d
    :parameters (?d ?df ?dt ?t1 ?t2)
    :precondition (and
        (disk ?d)
        (disk ?df)
        (disk ?dt)
        (on-top ?d ?t1)
        (on-top ?dt ?t2)
        (tower ?t1)
        (tower ?t2)
        (on-disk ?d ?df)
        (smaller ?d ?dt)
        )
    :effect (and
        (on-top ?df ?t1)
        (not (on-top ?d ?t1))
        (on-top ?d ?t2)
        (on-disk ?d ?dt)
        (not (on-top ?dt ?t2))
        (not (on-disk ?d ?df))
        )
)
```

3.1.3 Moving a disk on an empty tower to another empty tower

```
; move disk from tower to tower
; ?d is the disk to move
; ?t1 is the tower ?d is on
; ?t2 is the tower to move ?d to
(:action move-t-to-t
    :parameters (?d ?t1 ?t2)
    :precondition (and
        (disk ?d)
        (tower ?t1)
        (tower ?t2)
        (empty ?t2)
        (on-tower ?d ?t1)
        (on-top ?d ?t1)
        )
    :effect (and
        (empty ?t1)
```

```
        (not (empty ?t2))
        (on-tower ?d ?t2)
        (not (on-tower ?d ?t1))
        (on-top ?d ?t2)
        (not (on-top ?d ?t1))
        )
)
```

3.1.4 Moving a disk on an empty tower to a disk

```
; move disk from tower to disk
; ?d is the disk to move
; ?dt is the disk to move ?d to
; ?t1 is the tower ?d is on
; ?t2 is the tower ?dt is on
(:action move-t-to-d
    :parameters (?d ?dt ?t1 ?t2)
    :precondition (and
        (disk ?d)
        (disk ?dt)
        (tower ?t1)
        (tower ?t2)
        (on-top ?d ?t1)
        (on-tower ?d ?t1)
        (on-top ?dt ?t2)
        (smaller ?d ?dt)
        )
    :effect (and
        (not (on-top ?d ?t1))
        (on-top ?d ?t2)
        (not (on-tower ?d ?t1))
        (empty ?t1)
        (on-disk ?d ?dt)
        (not (on-top ?dt ?t2))
    )
)
```

3.2 The problem

Coding the problem is very easy. Using the predicates we know, we give the necessary information of the initial state and of the goal state:

```
(define (problem hanoi-problem) (:domain hanoi-domain)
(:objects
    tower1 tower2 tower3
    disk1 disk2 disk3
)

(:init
    (tower tower1)
    (tower tower2)
    (tower tower3)
    (disk disk1)
    (on-disk disk1 disk2)
    (smaller disk1 disk2)
    (smaller disk1 disk3)
```

```
    (disk disk2)
    (on-disk disk2 disk3)
    (smaller disk2 disk3)
    (disk disk3)
    (on-tower disk3 tower1)
    (not (empty tower1))
    (empty tower2)
    (empty tower3)
    (on-top disk1 tower1)
)

(:goal (and
    (empty tower1)
    (empty tower2)
    (on-top disk1 tower3)
    (on-disk disk1 disk2)
    (on-disk disk2 disk3)
    (on-tower disk3 tower3)
    )
)
)
```

3.3 The result

To run the problem, we will be using the Fast Downward planning system. After installing it from here, we can run the following command:

```
./fast-downward.py hanoi-domain.pddl hanoi-problem.pddl --heuristic "
    h=ff()" --search "astar(h)"
```

Which will give us the actions needed to solve the problem, as well as the cost (number of moves\actions):

```
(move-d-to-t disk1 disk2 tower1 tower3)
(move-d-to-t disk2 disk3 tower1 tower2)
(move-t-to-d disk1 disk2 tower3 tower2)
(move-t-to-t disk3 tower1 tower3)
(move-d-to-t disk1 disk2 tower2 tower1)
(move-t-to-d disk2 disk3 tower2 tower3)
(move-t-to-d disk1 disk2 tower1 tower3)
; cost = 7 (unit cost)
```

Note that the cost is indeed 7 - the minimum cost of solving the problem.

4 Coding the generalized problem

To code the problem for an arbitrary number of disks, we would need to add additional predicates in the *init* and *goal* sections of the problem file. Rather than changing the code for each number of disks, I coded a Python script that gets the number of disks as a parameter and quite literally writes the PDDL code to a file. The majority of the code is the same regardless of the number of disks and was thus hard coded, while some code depends on the number of disks (e.g. the *init* section will have more occurrences of the *smaller* and *on-disk* predicates, while the *goal* section will have more occurrences of the *on-disk* predicate). The source code can be found here.

hanoi.py is the python script (main file). Run it with:

```
python hanoi.py --disks n
```

where n is the number of disks.

If omitted, the number of disks defaults to 3.

It will create a domain PDDL file, problem PDDL file and then run them and output it to a file "sas_plan".

The folder "Three Disks" contains the generated files by running "python hanoi.py".

5 Conclusion

This is how one would write PDDL code to solve problems. This language makes problem-solving easier and faster, as the programmer must only provide the legal actions, the initial state and the goal state.

6 Bibliography

https://www.mathsisfun.com/games/towerofhanoi.html
https://stackoverflow.com/
https://stackabuse.com/executing-shell-commands-with-python/
https://realpython.com/
https://docs.python.org/3/library/sys.html
https://snakify.org/
https://machinelearningmastery.com/command-line-arguments-for-your-python-script/
https://www.w3schools.com/python/
https://towardsdatascience.com/a-simple-guide-to-command-line-arguments-with-argparse-6824c30ab1c3