PLANNING

Academic year 2020-2021

We thank **Chiara Ghidini** and **Luciano Serafini** from whom
we borrowed the LaTeX style.

ï»¿

A goal without a plan is just a wish.

ANTOINE DE SAINT-EXUPERY

Chapter 1

Planning

1.1 Easy problems

These problems are simple modeling problems stressing that

1. the initial state,
2. the preconditions, and
3. the goal state

are sets of positive literals.

Exercise 1.1. 

Consider a scenario in which you have a set $S = \{l_1, \ldots, l_n\}$ of locations some of which are connected and some which are not. You have to specify the following domain/problem(s) in PDDL. Assuming that $n = 4$, and that l_i is connected to l_{i+1} with $i \in \{1, \ldots, n-1\}$, is there a path that starts at l_1 and ends at l_n? Under similar assumptions, try to write the problem file for $n = 6$, and then, still with $n = 6$, provide a problem for which a plan is not possible by severing the connection between, e.g., location l_4 and location l_5.

Solution.

We write the domain description in a file named `BasicMove.pddl`*:*

```
(define (domain BasicMove)
    (:requirements :strips)

    (:predicates (at ?l)
                 (conn ?l1 ?l2)
    )

    (:action move
        :parameters (?from ?to)
        :precondition (and (at ?from) (conn ?from ?to))
        :effect (and (at ?to) (not (at ?from)))
    )
)
```

An example of a problem instance with four locations is given below (file `BasicMove-four.pddl`):

```
(define (problem BasicMove-four)
    (:domain BasicMove)

    (:objects
        l1 l2 l3 l4
    )

    (:init
        (at l1)
        (conn l1 l2) (conn l2 l3) (conn l3 l4)
    )

    (:goal (at l4))
)
```

An example of a problem instance with six locations is given below (file `BasicMove-six-sat`). Notice that the domain does not change, only the number of objects (locations) is increased, and correspondingly further facts in the `:init` clause are added.

```
(define (problem BasicMove-six-sat)
    (:domain BasicMove)
```

```
    (:objects
        l1 l2 l3 l4 l5 l6
    )

    (:init
        (at l1)
        (conn l1 l2) (conn l2 l3) (conn l3 l4)
        (conn l4 l5) (conn l5 l6)
    )

    (:goal (at l6))
)
```

Another example of a problem instance with six locations is given below (file `BasicMove-six-unsat`*), but this time one of the connections (the one between l_4 and l_5) is severed. Again the domain does not change, but the facts in the* `:init` *clause are different (*`(conn l4 l5)` *does not appear any more).*

```
(define (problem BasicMove-six-unsat)
    (:domain BasicMove)

    (:objects
        l1 l2 l3 l4 l5 l6
    )

    (:init
        (at l1)
        (conn l1 l2) (conn l2 l3) (conn l3 l4)
        (conn l5 l6)
    )

    (:goal (at l6))

)
```

Exercise 1.2. ☼ ✍

Consider a scenario in which you have a set $S = \{l_1, \ldots, l_n\}$ of locations some of which are connected and some which are not. Some of the locations

are also obstructed by obstacles. You have to solve the following problem: Assuming that $n = 4$, and that l_i is connected to l_{i+1} with $i \in \{1, \ldots, n-1\}$, and that all the even locations are obstructed, is there a path that starts at l_1 and ends in l_n?

Solution.

We write the domain description in a file named `BasicMoveWO.pddl`*:*

```
(define (domain BasicMoveWO)
    (:requirements :strips)

    (:predicates (at ?l)
                 (unobs ?l)
                 (conn ?l1 ?l2)
    )

    (:action move
        :parameters (?from ?to)
        :precondition (and (at ?from) (conn ?from ?to) (unobs ?to))
        :effect (and (at ?to) (not (at ?from)))
    )
)
```

An example of a problem instance with four locations is given below (file `BasicMoveWO-four`*):*

```
(define (problem BasicMoveWO-four)
    (:domain BasicMoveWO)

    (:objects
        l1 l2 l3 l4
    )

    (:init
        (at l1)
        (conn l1 l2) (conn l2 l3) (conn l3 l4)
        (unobs l1)(unobs l3)
    )
```

```
    (:goal (at l4))
)
```

Exercise 1.3. ☼ ✍

Consider a scenario in which you have a set $S = \{l_1, \ldots, l_n\}$ of locations some of which are connected and some which are not. Some locations are also obstructed by obstacles and you do not wish to visit locations more than once. You have to specify the following domain/problem in PDDL. Assuming that $n = 4$, and that l_i is connected to l_{i+1} ($i = 1, \ldots, n-1$), that all the even locations are obstructed, is there a path that starts at l_1 and ends at l_n?

Solution.

We write the domain description in a file named `BasicMoveWOU.pddl`*:*

```
(define (domain BasicMoveWOU)
    (:requirements :strips)

    (:predicates (at ?l)
                 (unobs ?l)
                 (unvis ?l)
                 (conn ?l1 ?l2)
    )

    (:action move
        :parameters (?from ?to)
        :precondition (and (at ?from) (conn ?from ?to)
                           (unobs ?to) (unvis ?to))
        :effect (and (at ?to) (not (at ?from)) (not (unvis ?to)))
    )
)
```

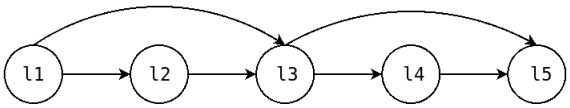
An example of a problem instance with four locations is given below (file `BasicMoveWOU-four-unsat`*):*

```
(define (problem BasicMoveWOU-four-unsat)
    (:domain BasicMoveWOU)
```

```
    (:objects
        l1 l2 l3 l4
    )

    (:init
        (at l1)
        (conn l1 l2) (conn l2 l3) (conn l3 l4)
        (unobs l1) (unobs l3)
        (unvis l2) (unvis l3) (unvis l4)
    )

    (:goal (at l4))

)
```

Exercise 1.4. ☀ ✍

Consider the same scenario as in the previous exercise but assume that $n = 5$ *and the connections between locations are given by the following directed graph:*



Provide the problem file in PDDL to check whether there a path that starts at l_1 *and ends at* l_5*.*

Solution.

The domain description is `BasicMoveWOU.pddl`*. The problem description becomes (*`BasicMoveWOU-graph5-sat`*):*

```
(define (problem BasicMoveWOU-graph5-sat)
    (:domain BasicMoveWOU)

    (:objects
        l1 l2 l3 l4 l5
    )

    (:init
```

```
        (at l1)
        (conn l1 l2) (conn l2 l3) (conn l3 l4) (conn l4 l5)
        (conn l1 l3) (conn l3 l5)
        (unobs l1) (unobs l3) (unobs l5)
        (unvis l2) (unvis l3) (unvis l4) (unvis l5)
    )

    (:goal (at l5))
)
```

Exercise 1.5. ☼ ✍

Consider a scenario in which you have a set $S = \{l_1, \ldots, l_n\}$ of locations some of which are connected and some which are not. Some locations are also obstructed by obstacles and you do not wish to visit locations more than once. Obstructed locations can be cleared of obstacles. You have to specify the following domain/problem in PDDL. Assuming that $n = 4$, and that l_i is connected to l_{i+1} with $i \in \{1, \ldots, n-1\}$, that all the even locations are obstructed, is there a plan that leads from l_1 to l_n?

Solution.

We write the domain description in a file named `BasicMoveClearWOU.pddl`*:*

```
(define (domain BasicMoveClearWOU)
    (:requirements :strips)

    (:predicates (at ?l)
                 (unobs ?l)
                 (obs ?l)
                 (unvis ?l)
                 (conn ?l1 ?l2)
    )

    (:action move
        :parameters (?from ?to)
        :precondition (and (at ?from) (conn ?from ?to)
                           (unobs ?to) (unvis ?to))
        :effect (and (at ?to) (not (at ?from)) (not (unvis ?to)))
    )
```

```
    (:action clear
        :parameters (?l)
        :precondition (obs ?l)
        :effect (and (unobs ?l) (not (obs ?l)))
    )
)
```

Notice the addition of the action `clear` *and the predicate* `obs` *which is the* dual *of* `unobs`*. Definition of both predicates is required to ensure that actions preconditions are sets of positive literals. The way operator* `clear` *is defined so as to maintain coherency between* `obs` *and* `unobs`*. An example of a problem instance with four locations is given below (file* `BasicMoveClearWOU-four-sat`*):*

```
(define (problem BasicMoveClearWOU-four-sat)
    (:domain BasicMoveClearWOU)

    (:objects
        l1 l2 l3 l4
    )

    (:init
        (at l1)
        (conn l1 l2) (conn l2 l3) (conn l3 l4)
        (unobs l1) (unobs l3)
        (obs l2) (obs l4)
        (unvis l2) (unvis l3) (unvis l4)
    )

    (:goal (at l4))
)
```

Notice that the initial state defines `obs` *and* `unobs` *coherently.*

1.2 Medium problems

These problems show the interactions between initial and/or goal states and/or actions.

Exercise 1.6. ☁ ✍

Consider a scenario in which you have a set $S = \{l_1, \ldots, l_n\}$ of locations some of which are connected and some of which are not. You have to specify the following domain/problem(s) in PDDL. ssuming that $n = 4$, and that l_i is connected to l_{i+1} with $i \in \{1, \ldots, n-1\}$ and that you want to visit all locations exactly once except for the initial location, is there a tour that starts at l_1 and ends in l_1? Consider also a different problem where a connection between l_4 and l_1 exists.

Solution.

We write the domain description in a file named `MoveUnvisVis.pddl`*:*

```
(define (domain MoveUnvisVis)
    (:requirements :strips)

    (:predicates (at ?l)
                 (conn ?l1 ?l2)
                 (vis ?l)
                 (unvis ?l)
    )

    (:action move
        :parameters (?from ?to)
        :precondition (and (at ?from) (conn ?from ?to) (unvis ?to))
        :effect (and (at ?to) (not (at ?from))
                     (not (unvis ?to)) (vis ?to))
    )

)
```

The problem corresponding to the description in the exercise is the following (file `MoveVisUnvis-four-unsat.pddl`*):*

```
(define (problem MoveUnvisVis-four-unsat)
    (:domain MoveUnvisVis)

    (:objects
        l1 l2 l3 l4
```

```
    )

    (:init
        (at l1)
        (conn l1 l2) (conn l2 l3) (conn l3 l4)
        (unvis l1) (unvis l2) (unvis l3) (unvis l4)
     )

    (:goal
        (and (at l1) (vis l1) (vis l2) (vis l3) (vis l4))
    )
)
```

Notice that `(vis l1)` *in the initial state is necessary for n=1. This problem domain does not admit a solution, because it is not possible from* l_4 *back to* l_1 *without passing over a location that was not visited. Adding a connection from* $l4$ *back to* $l1$ *guarantees that a solution can be found:*

```
(define (problem MoveUnvisVis-four-sat)
    (:domain MoveUnvisVis)

    (:objects
        l1 l2 l3 l4
    )

    (:init
        (at l1)
        (conn l1 l2) (conn l2 l3) (conn l3 l4) (conn l4 l1)
        (unvis l1) (unvis l2) (unvis l3) (unvis l4)
     )

    (:goal
        (and (at l1) (vis l1) (vis l2) (vis l3) (vis l4))
    )
)
```

Exercise 1.7. ☁ ✍

Consider a scenario in which you have a set $S = \{l_1, \ldots, l_n\}$ *of locations connected by roads, by bridges or not connected at all. You have to specify*

the following domain/problem in PDDL. There are $n = 4$ locations, for $i \in \{1, \ldots, 2\}$, location l_i is connected to l_{i+1} by a bridge, and for $i \in \{1, \ldots, 3\}$ location li is connected to $li + 1$ by a road; also, in order to take a bridge, the agent must come from a road, and the goal must be reached through a bridge. Finally all locations should be visited only once except for the initial location. Is there a tour that starts at $l1$ and ends in $l1$? Try to amend the description in order to make a plan possible.

Solution.

The domain is defined as follows (file `MoveLBridgesFroads.pddl`):

```
(define (domain MoveLBridgesFroads)
    (:requirements :strips)

    (:predicates (at ?l)
                 (vis ?l)
                 (unvis ?l)
                 (conn_road ?l1 ?l2)
                 (conn_bridge ?l1 ?l2)
                 (LBridge)
                 (FRoad)
    )

    (:action move_road
        :parameters (?from ?to)
        :precondition (and (at ?from) (conn_road ?from ?to) (unvis ?to))
        :effect (and (at ?to) (not (at ?from))
                     (not (unvis ?to)) (vis ?to)
                     (FRoad) (not (LBridge)))
    )

    (:action move_bridge
        :parameters (?from ?to)
        :precondition (and (at ?from) (conn_bridge ?from ?to) (unvis ?to))
        :effect (and (at ?to) (not (at ?from))
                     (not (unvis ?to)) (vis ?to)
                     (LBridge) (not (FRoad)))
    )
```

```
)
```

The problem corresponding to the description in the exercise is the following (file `MoveLBridgesFRoads-four-unsat.pddl`*):*

```
(define (problem MoveLBridgesFroads-four-unsat)
    (:domain MoveLBridgesFroads)

    (:objects
        l1 l2 l3 l4
    )

    (:init
        (at l1)
        (conn_bridge l1 l2) (conn_bridge l2 l3)
        (conn_road l1 l2) (conn_road l2 l3) (conn_road l3 l4)
        (unvis l1) (unvis l2) (unvis l3) (unvis l4)
     )

    (:goal
        (and (at l1)
             (vis l1) (vis l2) (vis l3) (vis l4)
             (LBridge))
    )
)
```

In order to make a plan possible there must be a bridge that leaves l_4 *in order to reach back to* l_1*. The following problem admits a solution (file* `MoveLBridgesFRoads-four-sat.pddl`*):*

```
(define (problem MoveLBridgesFroads-four-sat)
    (:domain MoveLBridgesFroads)

    (:objects
        l1 l2 l3 l4
    )

    (:init
        (at l1)
```

```
        (conn_bridge l1 l2) (conn_bridge l2 l3) (conn_bridge l4 l1)
        (conn_road l1 l2) (conn_road l2 l3) (conn_road l3 l4)
        (unvis l1) (unvis l2) (unvis l3) (unvis l4)
    )

    (:goal
        (and (at l1)
             (vis l1) (vis l2) (vis l3) (vis l4)
             (LBridge))
    )
)
```

Exercise 1.8. ☁ ✍

Consider a scenario in which you have a set $S = \{l_1, \ldots, l_n\}$ of locations, and a set $A = \{a_1, \ldots, a_m\}$ of agents. Some of the locations are connected by roads, some by bridges and some are not connected. Some agent can use both bridges and roads, while some other agents can use either bridges or roads (exclusively). Using PDDL, specify a scenario where:

- *there are two agents $m = 2$ and four locations, $n = 4$;*
- *l_i is connected to l_{i+1} with $i \in \{1, \ldots, 3\}$ by bridges;*
- *l_i is connected to l_{i+1} ($i = 1, \ldots, 3$) by roads;*
- *agent a_1 can use only roads while agent a_2 can use both roads and bridges.*
- *initially $a1$ is in $l1$ and $a2$ in $l2$.*

The goal is to have all locations visited by some agent.

Solution.

The domain is defined as follows (file `MoveAgentVBR.pddl`):

```
(define (domain MoveAgentsVBR)
    (:requirements :strips)

    (:predicates (loc ?l)
                 (agent ?a)
                 (at ?a ?l)
```

```
                (vis ?l)
                (uses_roads ?a)
                (uses_bridges ?a)
                (conn_road ?l1 ?l2)
                (conn_bridge ?l1 ?l2)
    )

    (:action move_road
        :parameters (?a ?from ?to)
        :precondition (and (agent ?a) (uses_roads ?a) (at ?a ?from)
                           (loc ?from) (loc ?to)
                           (conn_road ?from ?to))
        :effect (and (at ?a ?to) (not (at ?a ?from)) (vis ?to))
    )

    (:action move_bridge
        :parameters (?a ?from ?to)
        :precondition (and (agent ?a) (uses_bridges ?a) (at ?a ?from)
                           (loc ?from) (loc ?to)
                           (conn_bridge ?from ?to))
        :effect (and (at ?a ?to) (not (at ?a ?from)) (vis ?to))
    )
)
```

The problem corresponding to the description in the exercise is the following (file `MoveAgentsVBR-four-sat`*):*

```
(define (problem MoveAgentsVBR-four-sat)
    (:domain MoveAgentsVBR)

    (:objects
        a1 a2
        l1 l2 l3 l4
    )

    (:init
        (agent a1) (agent a2)
        (loc l1) (loc l2) (loc l3) (loc l4)
        (uses_roads a1)
```

```
        (uses_roads a2) (uses_bridges a2)
        (at a1 l1) (at a2 l2)
        (vis l1) (vis l2)
        (conn_road l1 l2) (conn_road l2 l3) (conn_road l3 l4)
        (conn_bridge l1 l2) (conn_bridge l2 l3) (conn_bridge l3 l4)
    )

    (:goal (and (vis l1) (vis l2) (vis l3) (vis l4)))


)
```

1.3 Hard problems

These problems are difficult to solve.

Exercise 1.9. ✍

The Tower of Hanoi is a mathematical game. It consists of three pegs, and a number of discs of different sizes which can slot onto any peg. The puzzle starts with the discs neatly stacked in order of size on one peg, smallest at the top. The objective of the game is to move the entire stack to another peg, obeying the following rules:

- *only one disc may be moved at a time;*
- *a disc can only be placed onto a larger disc, not necessarily of the size immediately larger than its size (e.g., the smallest disc may sit directly on the largest disc).*

You have to specify the domain in PDDL and provide a problem files for three and four discs. How does the sice of the PDDL encoding grow with respect to the number of discs?

Solution.

The definition of the problem domain is quite lean. We define 3 predicates: unary predicate "clear", binary predicates "on" and "smaller". One action "move" is sufficient. The domain can be described as follows (file `Hanoi.pddl`*):*

```
(define (domain Hanoi)
    (:requirements :strips)

    (:predicates
        (clear ?x)
        (on ?x ?y)
        (smaller ?x ?y)
    )

    (:action move
        :parameters (?disc ?current ?dest)
        :precondition (and (clear ?disc) (clear ?dest)
                           (on ?disc ?current) (smaller ?disc ?dest))
        :effect (and (not (clear ?dest)) (not (on ?disc ?current))
                     (clear ?current) (on ?disc ?dest))
    )
)
```

Notice that there is no predicate to distinguish between pegs and discs: the planner will treat them as members of the same domain. The key for solving the puzzle is in the specification of the problem. In this, somewhat unintuively, we declare that pegs are objects that are always smaller than discs. This ensures that we can always move a disc to a peg.

```
(define (problem Hanoi-3pegs3discs)
    (:domain Hanoi)

    (:objects d1 d2 d3 peg1 peg2 peg3)

    (:init
        (clear d1) (clear peg2) (clear peg3)
        (on d1 d2) (on d2 d3) (on d3 peg1)
        (smaller d1 d2) (smaller d2 d3) (smaller d1 d3)
        (smaller d1 peg1) (smaller d1 peg2) (smaller d1 peg3)
        (smaller d2 peg1) (smaller d2 peg2) (smaller d2 peg3)
        (smaller d3 peg1) (smaller d3 peg2) (smaller d3 peg3)
    )
```

```
    (:goal
        (and (clear peg1) (clear peg2) (clear d1)
             (on d1 d2) (on d2 d3) (on d3 peg3))
    )
)
```

We can generalize the encoding to n discs. The specification for 4 discs is the following (`Hanoi-3pegs-4discs`)

```
(define (problem Hanoi-3pegs-4discs)
    (:domain Hanoi)

    (:objects d1 d2 d3 d4 peg1 peg2 peg3)

    (:init
        (clear d1) (clear peg2) (clear peg3)
        (on d1 d2) (on d2 d3) (on d3 d4) (on d4 peg1)
        (smaller d1 d2) (smaller d1 d3) (smaller d1 d4)
        (smaller d2 d3) (smaller d2 d4)
        (smaller d3 d4)
        (smaller d1 peg1) (smaller d1 peg2) (smaller d1 peg3)
        (smaller d2 peg1) (smaller d2 peg2) (smaller d2 peg3)
        (smaller d3 peg1) (smaller d3 peg2) (smaller d3 peg3)
        (smaller d4 peg1) (smaller d4 peg2) (smaller d4 peg3)
    )

    (:goal
        (and (clear peg1) (clear peg2) (clear d1)
             (on d1 d2) (on d2 d3) (on d3 d4) (on d4 peg3))
    )
)
```

*The encoding grows quadratically in the number of discs because we need to fully specify the relation `smaller` for the discs. Also, take into account that the **optimal** plan for the tower of Hanoi with three pegs has a number of moves which grows **exponentially** with the number of discs n — precisely, as $2^n - 1$. Indeed, `Hanoi-3pegs-3discs` can be solved in 7 steps, whereas `Hanoi-3pegs-4discs` takes 15 steps. It can be proved that a specialized solution algorithm can do no better than a general purpose planner, i.e., the*

problem provably requires exponential time to be solved. Incidentally, this also proves that general-purpose planning may be exponential in the size of the domain.

Exercise 1.10. ✍

Consider a set S of clauses. Formalize the problem of satisfying S as a planning problem in PDDL in the specific case of case $S = \{(v1, v2), (-v1, -v2)\}$.

Solution.

We introduce the unary predicates "var" to denote variables, "value" to denote thruth values, "clause" to denote clauses, "unvalued" to state that a variable has no valuation and "sat" to state that a clause is satisfied. We also introduce the binary predicate "evaluated" to state that a variable is assigned a truth value and the the ternary predicate "in" to state that a given variable appears in a clause either positively or negatively. Two actions "assignv" — to assign values to variables — and "assignc" — to assign values to clauses — are sufficient. The domain is defined as follows (file `SAT.pddl`):

```
(define (domain SAT)
    (:requirements :strips)

    (:predicates    (var ?v)
                    (value ?tf)
                    (clause ?c)
                    (sat ?c)
                    (unvalued ?v)
                    (evaluated ?v ?tf)
                    (in ?v ?c ?tf)
    )

    (:action assignv
        :parameters (?v ?tf)
        :precondition (and (var ?v) (value ?tf) (unvalued ?v))
        :effect (and (evaluated ?v ?tf) (not (unvalued ?v)))
    )

    (:action assignc
```

```
        :parameters (?v ?tf ?c)
        :precondition (and (var ?v) (value ?tf) (clause ?c)
                           (evaluated ?v ?tf) (in ?v ?c ?tf))
        :effect ((sat ?c))
    )
)
```

The problem instance must specify two objects for the truth values and then as many objects as there are variables and clauses in the formula. Objects must be declared of the right "sort" in the the initial condition, and then the structure of the formula must be specified. This exercise shows that the complexity of planning is at least NP-hard since you can encode Boolean satisfiability into a planning problem.

```
(define (problem SAT-2var-2cl)
    (:domain SAT)

    (:objects
        t f
        v1 v2
        c1 c2
    )

    (:init
        (value t) (value f)
        (var v1) (var v2)
        (clause c1) (clause c2)
        (in v1 c1 t) (in v2 c1 t)
        (in v1 c2 f) (in v2 c2 f)
        (unvalued v1) (unvalued v2)
    )

    (:goal (and (sat c1) (sat c2)))
)
```

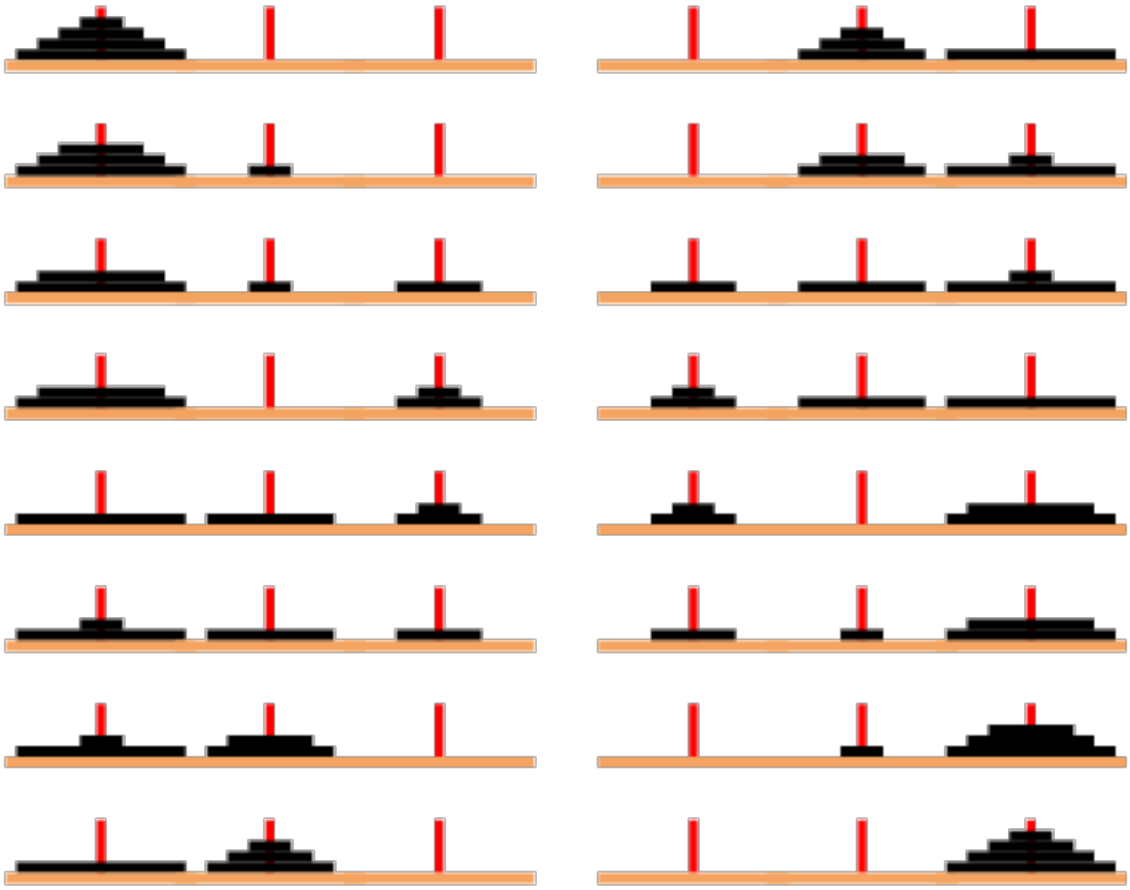


Figure 1.1: Solution of the tower of Hanoi puzzle with four discs. Initial configuration (top-left) and final configuration (bottom-right). The sequence of configurations is arranged top-down and then left-to-right.