Contents

CHAPTER 2
Discrete and Deterministic Planning

PDDL supports the expression of several different classes of planning problems. In this chapter and the next, we begin with the simplest class of planning problems, which are discrete, finite, and deterministic. This kind of planning is often called *classical* (to distinguish it from the many different extensions that have later followed).

In PDDL, the state of the world we model is described by the set of *facts* that are true in it. The actions in a plan cause state transformations, which alter the set of true facts according to the action's *effects*. Further, each action has a *precondition*, which determines in which states the action can be applied. The classical planning problem makes the following assumptions.

- The set of facts that may be true or false (or, in other words, the set of state variables) and the set of possible actions is finite.
- The models of actions available to the planner are correct and deterministic. This means that applying the same action in the same state always leads to the same result.
- The planner has complete knowledge of the initial state (there are no unknown facts), and the world is static, meaning that there are no changes to the state other than those caused by the planned actions.

These may appear to be highly unrealistic and limiting assumptions. Nevertheless, the classical planning model is useful for several reasons. First, since it provides a simplified, idealised setting, it is a good starting point for studying and developing automated planners. Many of the lessons learned and techniques discovered for classical planning have generalised to, or inspired adaptations of methods for, more complex planning problems.

Second, in many cases even the classical planning model is enough to express a sufficiently good approximation of a problem. The formal model of a planning problem is usually an abstraction of the real problem. Aspects of a problem that we know can be resolved when the plan is executed do not need to be considered at planning time. For example, if we plan to walk from one place to another, we do not plan ahead in detail every leg movement or placement of feet on the ground. The plan executor—whether it is a robot, control system, or human—has some known capabilities (or "skills"), and the actions of the planning model are what it can do using those skills. The same is true of surprises, whether they arise from incomplete knowledge of the state or from an incomplete model of the preconditions or effects of actions. In any realistic situation, the list of unexpected things that could happen—actions failing or having unexpected side-effects, or another actor interfering with the state while planning is done—is endless. But if we cannot make plans to avoid unexpected disruptions, nor plan in advance how to recover from them if they happen, then it is not useful to include them in the planning model.

In the next section we will introduce the basic building blocks for PDDL by way of a small number of examples of increasing complexity. We follow in Section 2.2 with a formal discussion of *plans*—the desired solution of a planning problem—and then discuss some further details of the syntax of the classical planning subset of PDDL in Section 2.3. We detail some more advanced modelling examples in Section 2.4, and finish with a discussion on the complexity of solving planning problems in Section 2.5.

2.1 DOMAIN AND PROBLEM DEFINITION

PDDL divides the definition of a planning problem into two parts: the *domain* defines the state variables (facts that may be true or false) and actions, while the *problem* defines initial state and the goal condition. Thus, a domain definition is a general model of the relevant aspects of the world in which we are planning, while the problem definition is a specific problem instance in this domain that specifies where we begin and what we must achieve. The domain and problem definitions are usually placed in two files, using the extension "`.pddl`". This is not mandated by the PDDL language, but many planners require it.

In the remainder of this section, we introduce the basic features of PDDL through a series of annotated examples.

2.1.1 PDDL: FIRST EXAMPLE

We begin with a very simple example (a kind of planning "hello world"). Imagine that we have a switch: the switch can be in one of two positions ("on" or "off") and the two actions we can take is to move it from one position to the other. Here is a complete definition of this domain in PDDL:

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

  (:predicates (switch_is_on)
               (switch_is_off))

  (:action switch_on
   :precondition (switch_is_off)
   :effect (and (switch_is_on)
                 (not (switch_is_off)))
   )

  (:action switch_off
   :precondition (switch_is_on)
   :effect (and (switch_is_off)
                (not (switch_is_on)))
   )
)
```

PDDL Example 1: A domain definition for the switch example.

We can make several observations about the language from this example: All PDDL expressions are enclosed in matching parentheses. The order of expressions within a pair of parentheses is prefix, meaning operator followed by arguments. For example, the conjunction that appears in the effects of the actions is written `(and <conjunct1> ... <conjunctN>)`. These syntactical conventions are inherited from the LISP programming language.

Most keywords start with a colon (`:`), but there are some exceptions (for example, `define`, `domain`, and the logical operators `and` and `not`). The remaining terms in this example are names (the name of the domain, and the names of predicates and actions). The PDDL version 1 specification [McDermott et al., 1998] states that valid names "are strings of characters beginning with a letter and containing letters, digits, hyphens (-) and underscores (_). Case is not significant". However, many planners will only accept a narrower set. Names made up of alphanumeric characters (a–z, 0–9) and underscores (_) should be acceptable to all planners. Note that names are case insensitive. Line breaks have no semantic meaning in PDDL. Whitespace in general is ignored, except that it is required to separate names, keywords, and other lexical elements where they appear in sequence without an intervening parenthesis.

As we have mentioned, different subsets of PDDL allow the specification of different classes of planning problems. The `:requirements`, section of the domain definition indicates which features of PDDL the domain uses, and thus, to some extent, what kind of planning problem it is. In this example, the keyword `:strips`, indicates that the domain is of the simplest form. The requirements specification is somewhat redundant—most planners will examine the domain definition itself to determine if it uses any PDDL features that they cannot deal with rather than rely on the requirements keywords. It is, however, good practice to always write a correct requirements specification.

The `:predicates` section of the domain definition contains the list of the model's state variables. These are binary variables, meaning they represent facts that are either true or false. In this small example, there are only two state variables and therefore only four possible states, which are shown in Figure 2.1. Actions define transitions between states. Each action has a precondition, which defines when the action is applicable, and an effect that defines what happens when the action is applied. In the `:strips` fragment, an action's precondition can be a single fact or a conjunction of facts. For example, in Figure 2.1, action `(switch_on)` is applicable only in states where the fact `(switch_is_off)` is true (the two states on the right). The effect is written as a conjunction of facts made true and facts made false. The latter are written as negated literals. Thus, the effect of, for example, the `switch_on` action is to make `(switch_is_on)` true and `(switch_is_off)` false. Note that if this action is applied to the state where both facts are already true (lower right in Figure 2.1) only one of them changes.

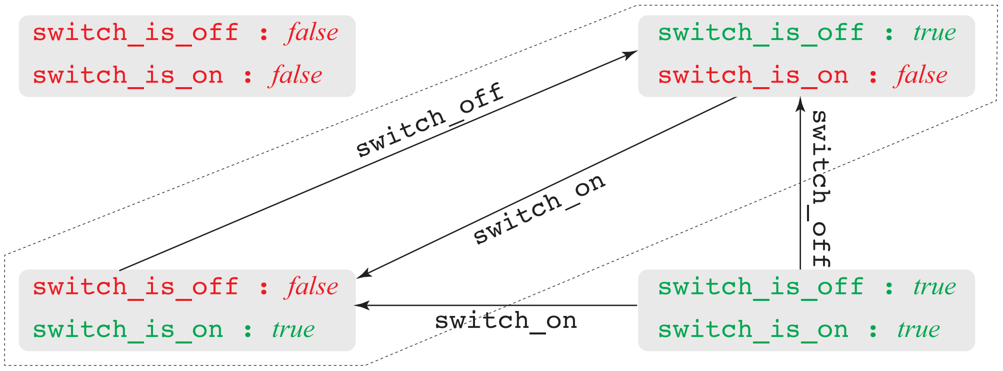

Figure 2.1: All possible world states of the model in the `switch` domain, Example 1, and the transitions between states caused by actions. The encircled states are the intended reachable part of the state space, i.e., those that correspond to actual possible states of the switch.

In this example, it is of course our intention that at any time exactly one of the facts `(switch_is_on)` and `(switch_is_off)` should be true, and the other false, since the switch can only be in one position. That is, of the four states in Figure 2.1 only two are “valid” (these two states are encircled with a dotted line). PDDL has no general mechanism for enforcing this kind of constraint on states (although it does allow for negative literals in action preconditions, as we will see later in this section, which allows us to eliminate these “spurious” states in this example with a modified model). However, the way in which we have written the two actions ensure that if we start from one of the states in the valid set, no action can take us out of it. If the initial state lies in this set, we refer to it as the *reachable* state space. A property that holds in all reachable states is often called a *state invariant*. Our task when defining a planning domain is to write it so that the reachable state space corresponds to that of our intended problem.

The PDDL problem definition specifies the initial situation and the goal condition. (There are more components to planning problems that will be introduced later in this section.) Here is an example problem definition for our simple switch domain.

```
(define (problem turn_it_off)
  (:domain switch)

  (:init
   (switch_is_on)
   )

  (:goal (switch_is_off))
)
```

PDDL Example 2: A problem definition for the switch example.

Some key observations we can make from this are the following. The problem, like the domain, is named. The problem definition also includes a reference to the domain that it is associated with, in the form `(:domain <name>)`. Note that, much like the `:requirements` specification, this information is somewhat redundant: most planners require as input two files, one with a domain definition and the other with a problem definition, and assume the two go together. It is, however, good practice to name the right domain in the problem definition.

The `(:init ...)` section defines the initial state of the problem instance, by listing all facts that are true in that state. Every fact that is not explicitly mentioned is assumed to be false initially. This means that in this example, the initial state is the lower left state in Figure 2.1. Note that the `:init` keyword is followed directly by a list of facts; there is no `and`, or additional parentheses.

The `:goal` section specifies a condition that must be satisfied at the end of a valid plan. The goal condition has the same form as an action precondition; for `:strips`, the form is either a single fact or a conjunction of facts. In this example, the goal is simply for the switch to be off. Note that the goal condition does not have to specify a unique state, or even a unique reachable state. As we shall soon see, however, more expressive planning formalisms relax the restriction on how we express a goal.

2.1.2 EXAMPLE: MODELLING THE KNIGHT'S TOUR

In this section we show how to model the Knight's Tour problem, described in Section 1.3.1. We will show how using *parameters* and *objects* we can write a PDDL domain and problem that is more compact and general. But first, let's look at how to formulate the Knight's move as an action.

Tool support: PDDL syntax checking

The VAL tool suite, available at https://github.com/KCL-Planning/VAL, includes a PDDL syntax checker, called `parser`. It is simple to use:

```
$ parser domain-file.pddl problem-file.pddl
```

The checker's output can be quite verbose, because it issues warnings for every item in the PDDL definition that implies a missing `:requirements` keyword.

Another useful tool is https://planning.domains. This is an on-line repository of PDDL domain and problem definitions, but includes also an editor with PDDL syntax highlighting and an option to run a planner.

```
(:action move_A8_to_B6
 :precondition (and (at_A8)
                    (not (visited_B6)))
 :effect (and (not (at_A8))
              (at_B6)
              (visited_B6))
 )
```

PDDL Example 3: A possible formulation of a Knight's move as an action.

The action shown above moves the Knight from one square A8 to square B6, which is a valid Knight's move. (We follow the convention of naming columns with letters and rows with numbers. Figure 1.3, on page 4, shows the naming scheme.) The action's precondition requires that the Knight is on the origin square, and that the destination square has not already been visited (recall that the aim of the puzzle is for the Knight to visit every square exactly once). Note that we use the negation of the predicate here: this is allowed in PDDL if the keyword `:negative-preconditions` is added to the `:requirements` section. The effect makes predicate `(at_A8)` false and predicate `(at_B6)` true, representing the fact that the location of the piece has changed; it also makes `(visited_B6)` true, so that the Knight won't return to this square later in the plan. There are 336 valid Knight's moves on a chess board. Thus, we could model the puzzle by writing 336 actions like this one, using 64 predicates for the possible current locations of the Knight and 64 to keep track of the squares visited. However, these actions are all very similar: they differ only by which two squares the Knight moves between.

Parameterising predicates and actions allows us to write a smaller number of predicates and actions—in the case of the Knight's tour, only one action—that are then *instantiated* with a collection of possible objects. The following is a parameterised version of the Knight's Tour domain:

```
(define (domain knights-tour)
  (:requirements :negative-preconditions)

  (:predicates
    (at ?square)
    (visited ?square)
    (valid_move ?square_from ?square_to)
   )

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

PDDL Example 4: A parameterised domain formulation for the Knight's Tour Puzzle.

We have included the PDDL requirement keyword `:negative-preconditions`, since the precondition of the `move` action uses a negated literal in its precondition.

The key new element in this domain is the use of parameters in the predicate and action declarations. Parameter symbols in PDDL must begin with a '`?`', and otherwise follow the rules for valid names. The parameters used in the declaration of a predicate must be all different. That is, we could not have written the predicate declaration `(valid_move ?square ?square)`. In the action definition, the list of parameters is written in the `:parameters` field. To remove any potential ambiguity in the action's meaning, literals in the preconditions and effects of an action can only use parameter symbols that are declared as parameters of the action. Note that these do not need to equal the parameters used in the predicate's declaration. We often refer to a parameterised action definition as an *action schema*.

The values that parameters can assume correspond to objects declared in the PDDL problem definition. Here is a portion of the problem definition that goes with the domain in Example 4:

```
(define (problem knights-tour-problem-8x8)
  (:domain knights-tour)

  (:objects
    A1 A2 A3 A4 A5 A6 A7 A8
    B1 B2 B3 B4 B5 B6 B7 B8
    ...
    H1 H2 H3 H4 H5 H6 H7 H8)

  (:init
   ; The Knight's starting square is arbitrary; here, we have
   ; chosen the upper right corner.
   (at A8)
   (visited A8)

   ; We have to list all valid moves:
   (valid_move A8 B6)
   (valid_move B6 A8)
   (valid_move A8 C7)
   (valid_move C7 A8)
   (valid_move B8 A6)
   (valid_move A6 B8)
   (valid_move B8 C6)
   ...
   )

  (:goal (and (visited A1)
              (visited A2)
              ...
              (visited H8)))
  )
```

PDDL Example 5: Part of the PDDL problem definition for the Knight's Tour Puzzle.

We now have an `:objects` section that lists all of the objects in the problem instance. In this example, the objects are squares of the chess board ("..." means we have abbreviated the listing for clarity). Here we also have our first example of comments in PDDL. A comment begins with a semicolon (`;`) and continues to the end of the line. (This comment syntax also derives from the fact that PDDL's syntax is based on that of LISP.) The `:init` and `:goal` sections, like before, list the facts that are true in the initial state and the conjunction of facts that must be true for the goal to be achieved, respectively. Note that they are lists of *ground* facts, meaning all predicate parameters must be instantiated with objects of the problem.

The predicate `valid_move` is an example of what is often called a *static predicate*, meaning a predicate that is not affected by any action. There is no special syntax to mark the predicate as static—it is only the fact that it does not appear in the `:effect` of any action that makes it so. Because a static predicate can not change, the instances of it that are true in the initial state are exactly those that will be true in any reachable state. Static predicates often play an important role in parameterised PDDL formulations by restricting the instantiation of action parameters. Consider the `move` action in the formulation of the Knight's Tour domain above: without `(valid_move ?from ?to)` in its precondition, this action would allow the Knight to jump between any pair of squares on the board. The presence of this literal in the precondition means that the action can be instantiated only with moves between those pairs that are explicitly listed as valid in the `:init` section of the problem. However, this also means that we have to list all 336 valid Knight's moves in the `:init` section.

A more compact formulation can be found by recognising the regularities among the valid moves. For example, for every valid move, the reverse move is also valid. Thus, we could write out only one direction for each move, and define an action `move_back` identical to `move` except for having the precondition `(valid_move ?to ?from)` with the two parameters reversed. But we can do better than that, by separating the two coordinates of each square into different parameters. Examples 6 and 7 below show the resulting domain and problem, respectively. They can also be found at `editor.planning.domains/pddl-book/knights_tour_2`.

```
(define (domain knights-tour)
  (:requirements :negative-preconditions)

  (:predicates
    (at ?col ?row)
    (visited ?col ?row)
    (diff_by_one ?x ?y)
    (diff_by_two ?x ?y)
   )

  ; Action move_2col_1row moves the Knight two steps along the horizontal
  ; axis (changing the column by 2) and one step along the vertical axis
  ; (changing the row by 1), while action move_2row_1col below does the
  ; opposite.

  (:action move_2col_1row
   :parameters (?from_col ?from_row ?to_col ?to_row)
   :precondition (and (at ?from_col ?from_row)
                      (diff_by_two ?from_col ?to_col) ; col +/- 2
                      (diff_by_one ?from_row ?to_row) ; row +/- 1
                      (not (visited ?to_col ?to_row)))
   :effect (and (not (at ?from_col ?from_row))
                (at ?to_col ?to_row)
                (visited ?to_col ?to_row))
   )

  (:action move_2row_1col
   :parameters (?from_col ?from_row ?to_col ?to_row)
   :precondition (and (at ?from_col ?from_row)
                      (diff_by_two ?from_row ?to_row) ; row +/- 2
                      (diff_by_one ?from_col ?to_col) ; col +/- 1
                      (not (visited ?to_col ?to_row)))
   :effect (and (not (at ?from_col ?from_row))
                (at ?to_col ?to_row)
                (visited ?to_col ?to_row))
   )

  )
```

PDDL Example 6: Domain definition for an alternative formulation of the Knight's Tour Puzzle.

```
(define (problem knights-tour-problem-8x8)
  (:domain knights-tour)

  ; Define a set of "numbers" 1..8:
  (:objects n1 n2 n3 n4 n5 n6 n7 n8)

  (:init
    ; Initial position of the Knight piece (upper left corner):
```

```
    (at n1 n8)
    (visited n1 n8)

    ; Here, we have to list all instances of the static
    ; predicates diff_by_two and diff_by_one:
    (diff_by_one n1 n2)
    (diff_by_one n2 n1)
    (diff_by_one n2 n3)
    (diff_by_one n3 n2)
    ...
    (diff_by_one n7 n8)
    (diff_by_one n8 n7)

    (diff_by_two n1 n3)
    (diff_by_two n3 n1)
    (diff_by_two n2 n4)
    (diff_by_two n4 n2)
    ...
   )

  (:goal (and (visited n1 n1)
              (visited n1 n2)
              ...
              (visited n8 n8)))
  )
```

PDDL Example 7: Part of the PDDL problem definition for the second formulation of the Knight's Tour Puzzle.

In this formulation, we use a pair of objects, one for row and one for column, to represent each square on the chess board. As a result, the predicates `at` and `visited` now have two parameters, and each version of the `move` action has four. The main simplification lies in the encoding of valid moves: because every valid Knight's move changes either the row by ± 2 and and the column by ± 1 or vice versa, we can encode these conditions on the two pairs of parameters (`?from_row` and `?to_row`, and `?from_col` and `?to_col`, respectively) separately. As can be seen in the new problem definition (Example 7), by using the same objects for positions on both axes we now only need eight, as well as fewer instances of the static predicates. The eight objects represent integers 1–8. (They are named `n1` through `n8` because names in PDDL cannot consist of digits only, and many planners require names to start with an alphabetic character.)

We will see the use of objects that represent a subset of integers, along with static predicates encoding arithmetic relations over them, again in the following sections.

These examples illustrate that in PDDL, as in any modelling language, there may be many possibilities to model the same problem or concept. In the formulation that explicitly lists all valid moves it is easier to modify what they are. For example, if we wanted to try to solve the Knight's tour on a "mutilated chess board" (one from which the two white corner squares are cut out) or a "doughnut chess board" (one from which the four squares at the centre are removed), this formulation could easily be adapted by simply removing the objects corresponding to the missing squares and any predicates that mention them from the problem definition. Thus, which formulation is preferable often depends on exactly what is the set of problem instances we want to solve.

2.1.3 EXAMPLE: LOGISTICS

In this section, we will formulate a small-scale practical logistics problem, studied by Kilby et al. [2015], in PDDL. We will use this example to introduce two more features of the PDDL language: our model will have objects of different *types*, and we will assign each action a *cost*, so that we can express the objective function.

The problem is the single-day linehaul problem of a consumer goods distributor, who uses a fleet of trucks to deliver goods requested by a set of customers. The goods are so-called "fast moving consumer goods"—in other words mainly fresh food—and customers are supermarkets and other retailers. Each truck starts from a central depot, visits a sequence of customers, and returns to the depot at the end. Because the distance from the depot to the area where the customers are is significant, each truck can make only one such tour in a day. Goods are divided into chilled and ambient, and each customer has requested a certain quantity of each type. Chilled goods can only be transported in refrigerated trucks, while ambient-temperature goods can be sent both on trucks with refrigeration and without. Besides being refrigerated or not, the available truck models also have different capacities and per-kilometer driving costs. The objective is to minimise the total cost which is the sum over all trucks of their per-kilometer cost times the distance travelled. On a given day, some trucks may not be used at all. Figure 2.2 shows a small example instance of the problem.

We will also see the encoding of numbers as objects, used for positions on the chess board in the second encoding of the Knight's tour above, again in the formulation of the linehaul problem. Here, however, the numbers represent capacities and quantities of goods instead of coordinates. The numeric planning fragment of PDDL, which we introduce in Chapter 4, allows for general numeric functions and expressions. However, because customer requests and truck capacities in this problem are integers (with relatively small ranges) we can also use the encoding of these as objects.

As mentioned in Section 1.3.2, practical logistics problems often have many more complexities, such as delivery time windows, multi-modal transport, pickup-and-delivery, combined

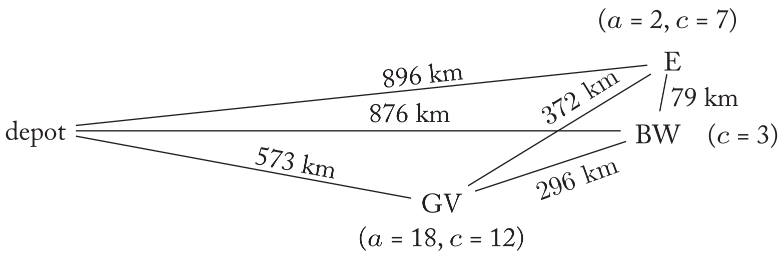

Figure 2.2: A small example instance of the Linehaul problem. The numbers next to customer locations show their demand for ambient and chilled goods.

routing and fleet selection, and so forth. Many of these can be formulated as variations of the model we present here, although some may require the use of a more expressive subset of PDDL. For example, to model time windows and other scheduling constraints we need the temporal planning subset, introduced in Chapter 5. In Chapters 3 and 4, we will introduce PDDL features that simplify modelling more complex objective functions.

Introducing Object Types

The following is a first step toward a domain definition. (The domain and example problem can be found at editor.planning.domains/pddl-book/linehaul_without_costs.) Here, we have not yet introduced action costs.

```
(define (domain linehaul_without_costs)
  (:requirements :strips :typing)

  (:types
    location truck quantity - object
    refrigerated_truck - truck
  )

  (:predicates
    (at ?t - truck ?l - location)
    (free_capacity ?t - truck ?q - quantity)
    (demand_chilled_goods ?l - location ?q - quantity)
    (demand_ambient_goods ?l - location ?q - quantity)
    (plus1 ?q1 ?q2 - quantity)
  )

  ;; The effect of the delivery action is to decrease demand at
  ;; ?l and free capacity of ?t by one.
  (:action deliver_ambient
    :parameters (?t - truck ?l - location
                 ?d ?d_less_one ?c ?c_less_one - quantity)
    :precondition (and (at ?t ?l)
                       (demand_ambient_goods ?l ?d)
                       (free_capacity ?t ?c)
                       (plus1 ?d_less_one ?d)   ; only true if x={?d,?c}, x > n0
                       (plus1 ?c_less_one ?c))   ; and x = x_less_one + 1
    :effect (and (not (demand_ambient_goods ?l ?d))
                 (demand_ambient_goods ?l ?d_less_one)
                 (not (free_capacity ?t ?c))
                 (free_capacity ?t ?c_less_one))
  )

  (:action deliver_chilled
    ;; Note type restriction on ?t: it must be a refrigerated truck.
    :parameters (?t - refrigerated_truck ?l - location
                 ?d ?d_less_one ?c ?c_less_one - quantity)
    :precondition (and (at ?t ?l)
                       (demand_chilled_goods ?l ?d)
                       (free_capacity ?t ?c)
                       (plus1 ?d_less_one ?d)   ; only true if x={?d,?c}, x > n0
                       (plus1 ?c_less_one ?c))   ; and x = x_less_one + 1
    :effect (and (not (demand_chilled_goods ?l ?d))
                 (demand_chilled_goods ?l ?d_less_one)
                 (not (free_capacity ?t ?c))
                 (free_capacity ?t ?c_less_one))
  )

  (:action drive
    :parameters (?t - truck ?from ?to - location)
    :precondition (at ?t ?from)
    :effect (and (not (at ?t ?from))
                 (at ?t ?to))
  )
)
```

There are several new elements in this domain definition.

- We added the keyword `:typing` to the requirements, to reflect that this domain uses objects of different types.
- We have added a `:types` section, in which we declare the names of the object types used.
- Finally, we have added a type specification to every parameter in the predicate and action declarations.

The following is the (abbreviated) problem definition for the example instance in Figure 2.2.

```
(define (problem linehaul-example)
  (:domain linehaul_without_costs)

  (:objects
    ADoubleRef - refrigerated_truck
    BDouble - truck
    depot GV E BW - location
    ;; Declare objects for integers, up to the largest quantity
    ;; that appears in the problem:
    n0 n1 n2 ... n40 - quantity
    )

  (:init
    (at ADoubleRef depot)
    (at BDouble depot)
    (free_capacity ADoubleRef n40)
    (free_capacity BDouble n34)
    (demand_chilled_goods GV n18)
    (demand_ambient_goods GV n12)
    (demand_chilled_goods E n7)
    (demand_ambient_goods E n2)
    (demand_chilled_goods BW n3)
    (demand_ambient_goods BW n0)
    (plus1 n0 n1)
    (plus1 n1 n2)
    ...
    (plus1 n39 n40)
    )

  (:goal (and (demand_chilled_goods GV n0)
              (demand_ambient_goods GV n0)
              (demand_chilled_goods E n0)
              (demand_ambient_goods E n0)
              (demand_chilled_goods BW n0)
              (demand_ambient_goods BW n0)
              (at ADoubleRef depot)
              (at BDouble depot)))
)
```

Note that the object names appearing in the `:objects` section are now typed. The two predicates `demand_ambient_goods` and `demand_chilled_goods` are used to track the remaining (i.e., as yet undelivered) demand for each goods category at each location. The `plus1` predicate expresses when the relation $X = Y + 1$ between pairs of quantity objects is true. In our example we specify that $\texttt{n1} = \texttt{n0} + 1, \ldots, \texttt{n40} = \texttt{n39} + 1$. The goal to deliver all requested goods is then expressed as the remaining demand being zero and all trucks being at the depot.

Object types in a PDDL domain form a hierarchy. The reserved word **`object`** denotes the top-level type, which encompasses all objects. In our linehaul problem, there are three types of objects of interest: locations (the depot and the customers), trucks, and quantities. Among trucks, we must further distinguish those that are refrigerated. In the domain above, we have declared **`location`**, **`truck`**, and **`quantity`** to all be subtypes of **`object`**, and furthermore we have declared **`refrigerated_truck`** to be a subtype of **`truck`**.

The role of types specifications is to restrict the instantiation of parameters to objects of the right type. That is, instead of creating all possible ground instances of an action schema by substituting the right number of objects, only those where the substituted objects' types are the same as, or a subtype of, the parameters' types, will be considered. Thus, in our domain definition, the `?t` parameter of the `deliver_ambient` action schema can be filled by an object of type **`truck`** or type **`refrigerated_truck`** (since the latter is a subtype), while the `?t` parameter of the `deliver_chilled` action schema can only be filled by an object of type **`refrigerated_truck`**. In this way, we enforce the constraint that chilled goods are only delivered by (and therefore only transported on) refrigerated trucks.

Type declarations for predicate parameters are essentially redundant, because wherever a predicate occurs, whether in an action schema, the initial state specification or the goal, the type of its arguments is given. Declaring more restrictive types for predicate parameters does not limit instantiation of action schemas (but will likely result in an error when passing the domain to a planner or PDDL validator). It may be viewed a good practice to type predicate parameters as a declaration of the domain designer's intent, but it can equally be regarded as an unnecessary source of errors.

The PDDL syntax for specifying the type of a parameter or object is by placing "`-` *<typename>*" after it. This can be done for parameters in predicate and action declarations in the domain, and in the `:objects` section of the problem definition. Note that the specification of the supertype of a type name in the `:types` section also uses the same syntax. A parameter or object that does not have a type specification belongs only to the top-level type `object` by default. Several parameters, or objects, can be typed with a single type specification by having multiple parameters to the left of the—symbol, as for example in the two delivery actions where the last four parameters are all specified to be of type `quantity`.

This is also an easy mistake to make: writing a declaration like

```
:parameters (?anything ?a_truck - truck)
```

intending the first parameter to be any type of object and only the second to be of the specified type. However, this declaration makes both parameters have type `truck`. If typed and untyped declarations are mixed in the same list (parameters, objects, or types), the untyped elements must be last in the list. Thus, the declaration

```
:parameters (?a_truck - truck ?anything)
```

does achieve the intended effect. Another way to accomplish the same is to use the reserved type name `object` for the parameter that is meant to be unconstrained:

```
:parameters (?anything - object ?a_truck - truck)
```

PDDL does not require that type names are declared before they are used in the `:types` section. Thus, another way of writing the type declaration section in our example is

```
(:types
  refrigerated_truck - truck
  location truck quantity
 )
```

This makes the last three types have the default supertype `object`. As is the case everywhere else in the PDDL language, line breaks have no special significance in the `:types` section.

A Formulation Without Types

Object types are not necessary to distinguish different kinds of objects. We have already seen that we can use *static* predicates—that is, predicates that are not modified by any action—to encode static relations between objects, such as, for example, the `valid_move` and `diff_by_two` predicates over locations to specify the valid Knight's moves, or `plus1` to specify the order of objects representing quantities. The effect of typing, which is to restrict instantiation of action schema parameters, can be achieved using static predicates of one argument. For example, we can write our linehaul domain as follows:

```
(define (domain linehaul_without_types)
  (:requirements :strips)
  (:predicates
    (is_a_truck ?obj)
    (is_a_refrigerated_truck ?obj)
    (is_a_location ?obj)
    (is_a_quantity ?obj)
    ... ; other predicates as above
  )

  (:action deliver_ambient
    :parameters (?truck ?loc ?d ?d_less_one ?c ?c_less_one)
    :precondition (and (is_a_truck ?truck)
                       (is_a_location ?loc)
                       (is_a_quantity ?d)
                       (is_a_quantity ?d_less_one)
                       (is_a_quantity ?c)
                       (is_a_quantity ?c_less_one)
    ... ; rest of the action schema as above
  )

  ... ; other action schemas rewritten in the same way
)
```

In the problem definition, instead of declaring object types, we specify the instances of the static predicates that are true:

```
(define (problem linehaul-example)
  (:domain linehaul_without_types)

  (:objects
    ADoubleRef
    BDouble
    depot GV E BW
    n0 n1 n2 ... n40
  )

  (:init
    (is_a_truck ADoubleRef)
    (is_a_refrigerated_truck ADoubleRef)
    (is_a_truck BDouble)
    (is_a_location depot)
    (is_a_location GV)
    (is_a_location E)
    (is_a_location BW)
    (is_a_quantity n0)
    (is_a_quantity n1)
    ... ; rest of initial state as above
    )

  ; goal is unchanged
  (:goal (and ...))
)
```

Note that since there is no defined supertype—subtype relationship here, we must explicitly state that ADoubleRef is both a truck and a refrigerated truck.

Since type information can also expressed with static predicates, what are the pros and cons of using PDDL's typing syntax? The main advantage is that it makes intended parameter and object types more explicit, and thereby can help avoiding modelling errors. For example, it is arguably easier to see that an action parameter is missing a type specification—particularly if we adopt the convention of explicitly typing every parameter, including using **object** where no type restriction is intended—than it is to notice that a typing predicate is missing from the action's precondition. Models written using types are also more compact, particularly if deep type hierarchies are used since the subtyping relationship implicitly declares an object to have all of its supertypes. On the other hand, static predicates can be more flexible in expressing properties of objects. For example, if we want to model an object that belongs to more than one type, or a type hierarchy that is not limited to single inheritance, it is unclear how to do this using PDDL's typing mechanism, or indeed whether it can be done at all.

Introducing Action Costs

The PDDL syntax for specifying action costs is actually a restricted case of the same syntax that is used for general numeric functions. (Number-valued functions that are not static are often called *fluents*.) We will cover all of it in Chapter 4 when we describe how to model numeric planning problems. Here, we will introduce only the subset that is permitted for action costs. The specification of action costs in a domain requires three steps:

1. declaring a special numeric function called total-cost, with no arguments;
2. adding to the effect of each action an expression that specifies how the action increases the total cost; and
3. adding a :metric specification to the problem definition.

Numeric functions are declared in a separate section of the domain definition called :functions. In addition to the total-cost function, we can also declare static functions, that can be used in the expression that calculates the cost of an action. The values of static functions must be enumerated in the initial state of the problem.

The linehaul domain with action costs is as follows. (We use the formulation with types, and show only the additions to the domain and problem in full. The complete domain and example problem can be found at `editor.planning.domains/pddl-book/linehaul_with_costs`.)

```
(define (domain linehaul_with_costs)
  (:requirements :strips :typing :action-costs)

  (:types
    ... ;; as above
  )

  (:predicates
    ... ;; as above
  )

  (:functions
    (distance ?l1 ?l2 - location) ; distance between locations
    (per_km_cost ?t - truck) ; per-kilometer cost of each truck
    (total-cost)
  )

  ;; actions deliver_ambient and deliver_chilled are unchanged

  (:action deliver_ambient
    ...
  )

  (:action deliver_chilled
    ...
  )

  (:action drive
    :parameters (?t - truck ?from ?to - location)
    :precondition (at ?t ?from)
    :effect (and (not (at ?t ?from))
                 (at ?t ?to)
                 (increase (total-cost)
                        (* (distance ?from ?to) (per_km_cost ?t))))
  )
)
```

Note the following changes.

- We have added the keyword `:action-costs` to the `:requirements` section.
- We have added a `:functions` section, declaring two static functions `distance` and `per_km_cost` and the `total-cost` function.
- We have added an `increase` effect to the driving action.

The general syntax of an increase effect is (`increase` *<function term> <expression>*). Because we are defining a domain with only action costs, not general numeric planning, the function term must be `total-cost`. The expression can be made up of numeric constants (integer or decimal), static functions, and the four standard arithmetic operators (`+`, `-`, `*` and `/`). Same as the logical operators, numeric expressions are written in prefix notation, i.e., with the operator first followed by both arguments. Addition and multiplication can only take two arguments, and there is no unary minus. Note that some planners may only accept more restricted forms of action cost specification, for example, only constants.

The problem definition is extended as follows:

```
(define (problem (linehaul_example)
  (:domain linehaul_with_costs)

  (:objects
    ... ;; as above
  )

  (:init
    ... ;; initial and static facts as above
    ;; specify distances between locations:
    (= (distance depot GV) 573)
    (= (distance depot E) 896)
    (= (distance depot BW) 876)
    (= (distance GV depot) 573)
    (= (distance GV E) 372)
    (= (distance GV BW) 296)
    (= (distance E depot) 896)
    (= (distance E GV) 372)
    (= (distance E BW) 79)
    (= (distance BW depot) 876)
    (= (distance BW GV) 296)
    (= (distance BW E) 79)
    ;; specify per-kilometer cost for the trucks:
    (= (per_km_cost ADoubleRef) 3.04)
    (= (per_km_cost BDouble) 2.59)
    ;; initialise total cost to zero:
    (= (total-cost) 0)
  )

  (:goal
    ... ; as above
  )

  (:metric minimize (total-cost))
)
```

The syntax for specifying the initial state value of a function, whether it is fluent or static, is `(= <function term> <constant>)`. Note that the problem must initialise the `(total-cost)` function to zero.

The final change is the addition of the `:metric` section, which specifies that the value of the `(total-cost)` function is to be minimised.

2.2 PLANS AND PLAN VALIDITY

The solution to a planning problem is a plan. In this section, we will define precisely what a plan is, and the conditions necessary for it to solve a planning problem. By doing this, we also give a precise semantics to the fragment of PDDL that we have presented so far.

Let D be a domain definition and P a problem definition for D. We will use the following notation to refer to the components of the domain and problem:

- $\text{types}(D)$ the set of type names mentioned in the `:types` section of D.
- $\text{predicates}(D)$ the set of predicates defined in D.
- $\text{actions}(D)$ the set of action schemas defined in D.
- For each action schema $a \in \text{actions}(D)$, $\text{name}(a)$ is the action's name and $\text{param}(a)$ is the sequence $x_1, \ldots, x_k$ of its parameters. For each parameter x_i, $\text{type-of}(x_i)$ is the type name that the ith parameter of the action is declared to have. If the parameter is declared without a type, $\text{type-of}(x_i) = \textbf{object}$. In the same way $\text{name}(p)$ and $\text{param}(p)$ denotes the name and parameters, respectively, of each predicate $p \in \text{predicates}(D)$.
- $\text{objects}(P)$ is the set of object names mentioned in the `:objects` section of P.

- init(P) is the set of ground facts listed in the `:init` section of P.
- goal(P) is the formula in the `:goal` section of P.

2.2.1 SEQUENTIAL PLANS

Informally, a plan is a sequence of actions. The actions that appear in a plan are names of ground instances of the action schemas in the domain. Formally, a plan is defined as follows.

Definition 2.1 Let $a \in \text{actions}(D)$ be an action schema, with $\text{param}(a) = x_1, \ldots, x_k$. Let $o_1, \ldots, o_k$ be a sequence of object names such that $o_i \in \text{objects}(P)$ for $i \in 1, \ldots, k$. Then $(\text{name}(a)\ \ o_1\ \ \ldots\ \ o_k)$ is a *ground action name*.

Definition 2.2 A *sequential plan* is a sequence of ground action names.

The following is a plan for the second version of Knight's Tour defined in Section 2.1.2 (Examples 6 and 7). It is the plan that is illustrated in Figure 1.3 in Chapter 1 (page 4).

```
(move_2row_1col n1 n8 n2 n6)
(move_2row_1col n2 n6 n3 n8)
(move_2col_1row n3 n8 n1 n7)
(move_2row_1col n1 n7 n2 n5)
(move_2row_1col n2 n5 n1 n3)
(move_2row_1col n1 n3 n2 n1)
(move_2col_1row n2 n1 n4 n2)
(move_2col_1row n4 n2 n6 n1)
(move_2col_1row n6 n1 n8 n2)
(move_2row_1col n8 n2 n7 n4)
(move_2row_1col n7 n4 n8 n6)
(move_2row_1col n8 n6 n7 n8)
(move_2col_1row n7 n8 n5 n7)
(move_2col_1row n5 n7 n7 n6)
(move_2row_1col n7 n6 n8 n8)
(move_2col_1row n8 n8 n6 n7)
(move_2col_1row n6 n7 n4 n8)
(move_2col_1row n4 n8 n2 n7)
(move_2row_1col n2 n7 n1 n5)
(move_2row_1col n1 n5 n2 n3)
(move_2row_1col n2 n3 n1 n1)
(move_2col_1row n1 n1 n3 n2)
(move_2col_1row n3 n2 n5 n1)
(move_2col_1row n5 n1 n7 n2)
```

```
(move_2row_1col n7 n2 n8 n4)
(move_2col_1row n8 n4 n6 n3)
(move_2row_1col n6 n3 n7 n1)
(move_2row_1col n7 n1 n8 n3)
(move_2col_1row n8 n3 n6 n2)
(move_2col_1row n6 n2 n8 n1)
(move_2row_1col n8 n1 n7 n3)
(move_2row_1col n7 n3 n8 n5)
(move_2row_1col n8 n5 n7 n7)
(move_2col_1row n7 n7 n5 n8)
(move_2col_1row n5 n8 n3 n7)
(move_2col_1row n3 n7 n1 n6)
(move_2row_1col n1 n6 n2 n8)
(move_2col_1row n2 n8 n4 n7)
(move_2col_1row n4 n7 n6 n8)
(move_2col_1row n6 n8 n8 n7)
(move_2row_1col n8 n7 n7 n5)
(move_2col_1row n7 n5 n5 n6)
(move_2row_1col n5 n6 n6 n4)
(move_2row_1col n6 n4 n5 n2)
(move_2col_1row n5 n2 n3 n1)
(move_2col_1row n3 n1 n1 n2)
(move_2row_1col n1 n2 n2 n4)
(move_2col_1row n2 n4 n4 n3)
(move_2col_1row n4 n3 n2 n2)
(move_2col_1row n2 n2 n4 n1)
(move_2row_1col n4 n1 n5 n3)
(move_2row_1col n5 n3 n6 n5)
(move_2col_1row n6 n5 n4 n4)
(move_2row_1col n4 n4 n3 n6)
(move_2col_1row n3 n6 n5 n5)
(move_2col_1row n5 n5 n3 n4)
(move_2row_1col n3 n4 n4 n6)
(move_2row_1col n4 n6 n5 n4)
(move_2row_1col n5 n4 n6 n6)
(move_2col_1row n6 n6 n4 n5)
(move_2row_1col n4 n5 n3 n3)
(move_2col_1row n3 n3 n1 n4)
(move_2col_1row n1 n4 n3 n5)
```

Tool support: Plan validation

The VAL tool suite (available at `https://github.com/KCL-Planning/VAL`) also includes a plan validator, called `validate`. It takes as input the domain and problem, and one or more plan files:

```
$ validate domain-file.pddl problem-file.pddl plan-file-1 ...
```

The plan file can be in several different formats, including one with time-stamped actions which is used for validation of plans in temporal and hybrid domains (see Chapters 5 and 6). For the classical fragment of PDDL, however, a plan is written simply as a list of ground action names, one per line, as shown in the example.

Another plan validator is INVAL (available at `https://github.com/patrikhaslum/INVAL`). It is used in the same way, but is restricted to validating non-temporal plans only. Since some areas of the specification of PDDL are ambiguous, the two validators do not always agree on whether a domain and problem is correct, or a plan is valid. Using both helps identify the areas where clarification may be needed.

Definition 2.2 states what a plan is. In the next two subsections, we define under what conditions a plan is *valid*, meaning that it is a solution to the planning problem posed by the domain and problem defintion.

Type Correctness

If the domain does not define any type names, all objects have the default type `object`. In this case, there is no restriction on which objects can be substituted for parameters of an action schema to make a ground action instance. If the domain uses typing, however, the first requirement for a plan to be valid is that the ground actions that make up the plan are instantiated with objects that are of the correct types for the action's parameters. To state this formally, we first have to define precisely the relation between objects and types. Because types in PDDL can form a hierarchy, there is a subtype—supertype relation over types, as well as a mapping from objects to the types they belong to. The combination of both can be expressed as a so-called "is-a" relation.

Definition 2.3 Let IS-A be a binary relation on objects$(P) \cup$ types$(D) \cup \{\textbf{object}\}$ such that:

(*i*) for all $t \in$ types(D), t IS-A **object**;

(*ii*) for all $o \in$ objects(P), o IS-A **object**;

(*iii*) if ... t_1 `...` `-` t_2 appears in the `:types` section of D, then t_1 IS-A t_2;

(*iv*) if ... o `...` `-` t appears in the `:objects` section of P, then o IS-A t;

(*v*) IS-A is transitive; and

(*vi*) IS-A is the smallest, w.r.t. inclusion, relation that satisfies conditions (*i*)–(*v*) above.

If the domain does not define any types, IS-A will simply relate each object to the top-level type name `object`.

Definition 2.4 Let $a \in \text{actions}(D)$ be an action schema with $\text{param}(a) = x_1, \dots, x_k$ and $n = (\text{name}(a)\ o_1\ \dots\ o_k)$ a ground action name. We say that n is *type correct* iff o_i IS-A $\text{type-of}(x_i)$ for $i = 1, \dots, k$. A plan is type correct iff every ground action name in it is type correct.

The use of object typing also places a number of requirements on the domain and problem definitions to ensure they are type correct. We will discuss these in more detail in Section 2.3, but in brief they are as follows. First, note that it is possible to write a PDDL domain definition so that the resulting IS-A relation is cyclic; this is not allowed. Second, for every type correct instantiation of an action schema the arguments of predicates in its precondition and effects must be of the correct type (and number) for those predicates. Finally, the initial state and goal specified in the problem, which are a list of ground literals and a ground formula, respectively, must also be type correct for the predicate parameters.

States and State Transitions

The actions of a plan cause changes to the state. If the plan is successful, the state reached at the end of it is one that satisfies the goal. In the discrete and deterministic fragment of PDDL, a state is defined by the set of facts that are true in it.

Definition 2.5 Let $p \in \text{predicates}(D)$ be a predicate, with $\text{param}(p) = x_1, \dots, x_k$. Let $o_1, \dots, o_k$ be a sequence of object names such that $o_i \in \text{objects}(P)$ and o_i IS-A $\text{type-of}(x_i)$ for $i \in 1, \dots, k$. Then $(\text{name}(p)\ o_1\ \dots\ o_k)$ is a *ground fact*.

Definition 2.6 Let F be the set of all ground facts. A *state* s is a subset of F.

We interpret a state $s \subseteq F$ as an assignment of a truth value to every ground fact $f \in F$: f is true if $f \in s$ and f is false if $f \notin s$. From this, s gives a truth value to every formula over ground facts, in the usual way: (`not` φ) is true iff φ is false, (`and` $\varphi_1\ \dots\ \varphi_n$) is true iff each of $\varphi_1, \dots, \varphi_n$ is true, and so on. The initial state defined by the problem, $\text{init}(P)$, is a state. If the `:init` section of the problem mentions a ground fact that is not a valid instantiation of a predicate of the domain, the problem is not well-formed.

Recall that a ground action name pairs the name of an action schema, $\text{name}(a)$, with a sequence of object names $o_1, \dots, o_k$. Each action schema in the domain must have a unique name, so we can find the corresponding schema definition from its name. From the ground action name, we can define a substitution mapping $\sigma = \{x_1 \to o_1, \ldots, x_k \to o_k\}$, where $\text{param}(a) = x_1, \ldots, x_k$. Applying this mapping to the precondition and effects of the action schema definition gives the precondition and effect of the ground action instance.

Definition 2.7 Let $a \in \text{actions}(D)$ be an action schema with $\text{param}(a) = x_1, \ldots, x_k$, $n = (\text{name}(a)\ o_1\ \ldots\ o_k)$ a (valid) ground action name and $\sigma = \{x_1 \to o_1, \ldots, x_k \to o_k\}$ the corresponding substitution mapping.

Let $\text{pre}(a)$ denote the `:precondition` formula of a. The *precondition of the ground action*, $\text{pre}(n)$, is the formula $\sigma(\text{pre}(a))$.

Let $\text{effect}(a)$ denote the `:effect` formula of a. The *add effect of the ground action*, $\text{add}(n)$, is the set of ground facts that appear positive (not negated) in the formula $\sigma(\text{effect}(a))$. The *delete effect of the ground action*, $\text{del}(n)$, is the set of ground facts that appear negative (within scope of `not`) in the formula $\sigma(\text{effect}(a))$.

Recall that the effect formula of an action schema must be a conjunction of literals, or a single literal. This restriction ensures that there is no ambiguity about what effects take place when the action is applied (there is no disjunction), and means that we treat the effect formula as a set of literals. The meaning of literals in the effect formula of an action schema is that applying the action makes the positive literals true and the negative literals false, while all facts not mentioned in the effect formula are unchanged. This meaning is captured by the definition of the add and delete effects of a ground action, respectively.

The sequence of ground action names that is the plan inductively defines a sequence of states, which are the states that result from applying each action to the previous state.

Definition 2.8 Let $n_1, \ldots, n_l$ be a plan. The *induced state sequence* $s_0, \ldots, s_l$ is defined inductively by $s_0 = \text{init}(P)$ and $s_i = (s_{i-1} \setminus \text{del}(n_i)) \cup \text{add}(n_i)$, for $i = 1, \ldots, l$.

Definition 2.9 Let $n_1, \ldots, n_l$ be a plan and $s_0, \ldots, s_l$ its induced state sequence. The plan is *executable* iff s_{i-1} satisfies $\text{pre}(n_i)$, for $i = 1, \ldots, l$. The plan is *valid* iff it is executable and s_l satisfies $\text{goal}(P)$.

Notice that in defining the state that results from applying a ground action n, we first remove the facts made false by the action, then add the facts it makes true. This means that if the effect formula of an action schema, under a particular instantiation, contains the same fact both positively and negatively, the net effect will be to make that fact true. However, the presence of the delete effect can still cause *action interference* when discussing non-sequential forms of plans, which is why it is important to distinguish this case from the case where the fact is only made true. Having the add effect take precedence over the delete effect, rather than the other way around, is just a choice made by the designers of PDDL. One should be aware, however, that not all planners implement this semantics; some may consider an action that both adds and deletes a fact to be invalid. From a problem modeller's perspective, the best course may be to avoid writing actions that do this.

We will not show step-by-step the execution of the entire plan for the Knight's Tour problem, but just the first action. The initial state is $s_0 = \text{init}(P) = \{$`(at n1 n8)`, `(visited n1 n8)`, `(diff_by_one n1 n2)`$, \ldots, \}$. The set of true instances of the predicates `diff_by_one` and `diff_by_two` will be the same in all states reached throughout executing the plan. As described in Section 2.1.2, these are static predicates, which do not appear in the effect of any action.

The first action in the plan is $n_1 =$`(move_2row_1col n1 n8 n2 n6)`. The precondition of this action is `(and (at n1 n8) (diff_by_two n8 n6) (diff_by_one n1 n2) (not (visited n2 n6)))`, which is satisfied in s_0. The delete effect is $\{$`(at n1 n8)`$\}$, and the add effect is $\{$`(at n2 n6)`, `(visited n2 n6)`$\}$. Thus, the state resulting from applying this action to s_0 is $s_1 = \{$`(at n2 n6)`, `(visited n1 n8)`, `(visited n2 n6)`, `(diff_by_one n1 n2)`$, \ldots, \}$.

2.2.2 NON-SEQUENTIAL PLANS

Above, we defined a plan as a sequence of actions. However, there are also less strictly ordered forms of plans. Lifting restrictions on the order of actions is important for scheduling a plan, since it gives flexibility to place the actions in time. However, we will not consider schedules in this chapter. In this section, we will give a brief overview of partially ordered plans. For more details, refer to the articles by Bäckström [1998] and Muise et al. [2016].

For simplicity, we will assume in the following that actions' preconditions are restricted to conjunctions of positive literals. Thus, for a ground action a, we may treat $\text{pre}(a)$ as a set of ground facts. We will also not distinguish between a ground action name and the corresponding ground action schema, as each one can be inferred from the other.

Definition 2.10 Two ground actions, a_1 and a_2, are said to be *non-interfering* iff $(\text{pre}(a_1) \cup \text{add}(a_1)) \cap \text{del}(a_2) = \emptyset$ and $(\text{pre}(a_2) \cup \text{add}(a_2)) \cap \text{del}(a_1) = \emptyset$.

That is, neither action deletes a precondition or add effect of the other. The significance of non-interference is that if both a_1 and a_2 are applicable in a state s, and the actions are non-interfering, then both can be applied one after the other, in either order, and the resulting state will be the same. This provides the foundation for relaxing the order of actions.

Non-interference does not imply concurrency. That two actions are non-interfering means they can be done in any order, but not necessarily that they can be done at the same time. (For example, cleaning the kitchen and cleaning the bathroom can be considered non-interfering actions, since neither prevents the other from being done later, or undoes the effect of the other. But to do them concurrently needs two cleaners.) Non-interference also does not imply that the order of actions a_1 and a_2 when they appear in sequence can be swapped. The first action may add a previously false fact that is required by the precondition of the second. Two actions are *commutative*, which is the term for “swappable” in this sense, iff they are non-interfering and $\text{add}(a_1) \cap \text{pre}(a_2) = \text{add}(a_2) \cap \text{pre}(a_1) = \emptyset$.

Definition 2.11 A *partially ordered plan*, $(T, \text{act}, \prec)$ consists of three elements: $T = \{1, \ldots, m\}$ is a set of *steps*, act is a function from T to ground action names, and $\prec$ is a strict partial order[1] on T.

The meaning is that T is the set of steps to be taken as part of the plan, $\text{act}(i)$ is the action that should be done at step i, and $\prec$ specifies the constraints on the order in which these steps can be done. If $\prec$ does not constrain the order of two steps i and j, we are free to choose the order in which to do them.

Validity of a partially ordered plan can be defined in two ways. Let’s start with the simple one: A *linearisation* of a partially ordered plan $(T, \text{act}, \prec)$ is the action sequence $\text{act}(i_1), \text{act}(i_2), \ldots,\ \text{act}(i_m)$, where $i_1, \ldots, i_m$ is a permutation of $T = \{1, \ldots, m\}$, such that $i_j \not\prec i_k$ for all $j > k$. In other words, a linearisation is a possible choice for how to place the actions corresponding to the steps of the plan in a sequence that is consistent with the ordering constraints. A partially ordered plan is valid iff every linearisation of it is a valid sequential plan (as defined in the previous section).

One may think that all unordered actions in a partially ordered plan must be non-interfering, but this is actually not the case. The plan can be valid even if $\text{act}(i)$ deletes a fact added by $\text{act}(j)$ and i and j are unordered, as long as no other action in the plan, or the goal, depends on this fact being true. However, for any two unordered steps i and j, we must have that $\text{pre}(\text{act}(i)) \cap \text{del}(\text{act}(j)) = \text{pre}(\text{act}(j)) \cap \text{del}(\text{act}(i)) = \emptyset$, since there exist linearisations with $\text{act}(i)$ immediately before $\text{act}(j)$ and linearisations with $\text{act}(j)$ immediately before $\text{act}(i)$, and if one action deletes a precondition of the other at least one of these linearisations will not be valid, since the second action will not be applicable in the state that results after the first.

The second way to define the validity of a partial-order plan rests on the notion of a *causal link*. A causal link is a triple (i, p, j), where i and j are steps in T and p a ground fact, such that $i \prec j$, $p \in \text{add}(\text{act}(i))$ and $p \in \text{pre}(\text{act}(j))$. It records an intention that the precondition p required by step j is achieved by step i. A step k with $p \in \text{del}(\text{act}(k))$ is a potential *threat* to the causal link (i, p, j). The link is *safe* from the threat of step k iff any one of the following conditions hold: (1) $k \prec i$; (2) $j \prec k$; or (3) there exists a step l with $p \in \text{add}(\text{act}(l))$ such that $k \prec l \prec j$. The link is *unthreatened* iff it is safe from every potential threat. If the link is unthreatened, then any step that deletes p either cannot appear between i and j in any linearisation, or there is another step (l) that re-establishes p and that must appear between the step that deleted it (k) and the step where it is needed (j). Note that condition (1) above is actually a special case of condition (3), with step i acting as the re-establishing step. To (re-)define plan validity, we need to extend the partially ordered plan with two “fake” steps, 0 and $m + 1$. These represent the initial state and the goal, respectively. For all $i \in T$, we have $0 \prec i \prec m + 1$. With slight abuse

[1]A strict partial order is a binary relation that is irreflexive, transitive, and anti-symmetric.

of notation, we say that $\mathrm{add}(\mathrm{act}(0)) = s_I$ and $\mathrm{pre}(\mathrm{act}(m+1)) = G$, even though these steps are not associated with any action. That is, step 0 adds all facts that are initially true, and step $m+1$ requires all goals to be true. The extended partial-order plan $(T \cup \{0, m+1\}, \mathrm{act}, \prec)$ is valid iff for every step $i \in T \cup \{0, m+1\}$ and every fact $p \in \mathrm{pre}(\mathrm{act}(i))$ there exists an unthreatened causal link (j, p, i). This condition is equivalent to the previous one, that all linearisations are valid. (The equivalence was shown by Nebel and Bäckström [1994], although they expressed it in quite different terms.)

Partially ordered plans are distinct from *partial-order planning*, which is a particular method of searching for a plan (that is, of turning a planning problem into a search space). Partial-order planning generates partially ordered plans, but it is equally possible to obtain a partially ordered plan by other means. In particular, a valid sequential plan can be converted into a valid partial-order plan. This process is known as *deordering* [Bäckström, 1998].

2.3 NOTES ON PDDL'S SYNTAX: THE STRIPS FRAGMENT

The examples presented so far in this chapter have introduced most of the basic features of the subset of PDDL known as "STRIPS", plus its extension to negated literals in action preconditions. (The use of `not` in action preconditions or goals is considered by many to not be within the STRIPS fragment of PDDL.) In this section, we will mention some features that we have not yet seen, and point out some restrictions that are not obvious, as well as some areas of potential ambiguity.

Lexical Elements: Valid Names, Case, and Whitespace

Owing to its origin in LISP, the lexical structure of PDDL is quite simple: it consists of parentheses, symbols (keywords, operators, names, and parameter names), and numeric literals. As we have mentioned, valid names in PDDL are defined to consist of alphanumeric characters, hyphens (`-`), and underscores (`_`), and must begin with a letter [McDermott et al., 1998]. However, many planners implement different, either more permissive or more restrictive, definitions. In particular, some planners may not accept names that include hyphens. It is reasonable to assume that reserved words (such as **`define`**, **`domain`**, **`problem`**, `and`, `not`, and others) cannot be used as names, although it has never been clearly specified.

Both keywords and names are case insensitive, so they can be written using any combination of upper and lower case. Whitespace is ignored, except where it serves to separate symbols. In particular, line breaks have no semantic meaning anywhere in PDDL. There is, however, one point of ambiguity: is space required to separate the hyphen in a type specification from the parameter or the type name? Some planners may accept, for example, `?t-truck`, `?t -truck`, and `?t - truck` as equivalent, all declaring a typed parameter `?t`, while other planners would read the first form as a parameter `?t-truck` with no type specified, the second as a syntax error (due to the symbol beginning with `-`), and only the third form in the intended way. Inserting spaces around the type specifier is the safer option.

Scope and Name Uniqueness

The first PDDL specification [McDermott et al., 1998] stated that within the scope of a domain and problem, all type, function, predicate, action, and object names must be unique. This of course means uniqueness within each category—that is, that we cannot have, for example, two different actions with the same name but different parameters—but it also means uniqness across all name categories; for example, we cannot have a type and an object with the same name. The only exception to this rule is that the domain and problem names may also be used as names within the domain or problem, because these names exist at different scopes. However, the same specification also states that "an atomic type name is just a timeless unary predicate, and may be used wherever such a predicate makes sense", thus suggesting that types are also implicitly declared as predicates.

While the name uniqueness rules have not been explicitly contradicted in any later PDDL version, they are also not strictly adhered to in existing planner, or even plan validator, implementations.

Disjunctive and Multiple Type Specifications

PDDL has a syntax for specifying a disjunction of types: if $t_1, \ldots, t_n$ are type names, then `(either` t_1 `...` t_n`)` also denotes a type, which is the union of $t_1, \ldots, t_n$. The interpretation of disjunctively typed parameters is straighforward: for example, `?transport - (either ship horse)` means that the parameter `?transport` can be instantiated by an object whose type is `ship` or `horse`, or any subtype of either of those. However, using a disjunctive type in an object declaration, or as the supertype when declaring a type, has no reasonable interpretation, since it either makes the definition nondeterministic, in the sense that we don't know what type the object is, or changes the meaning of `either` to something that is, arguably, its opposite.

The converse of disjunctive types is multiple inheritance, meaning types having multiple immediate supertypes in the hierarchy, or objects belonging to multiple types. For example, suppose we wanted to declare that `horse - vehicle` *and* `horse - herbivore`? Whether or not such multiple specifications for the same type or object can appear in type or object declarations is unknown—it is neither permitted nor ruled out by any of the existing documents on PDDL.

Disjunctive as well as multiple type specifications do not have wide-spread support in planner implementations, and are best avoided. As illustrated in Section 2.1.3, one can always use static predicates to capture object properties that are not expressed by their type.

Implicit Declarations

PDDL allows objects, optionally with types, to be declared in the domain definition as well as in the problem definition. This is done in a section marked by the keyword `:constants`, which follows the same form as the `:objects` section in the problem.

However, due to common use in collections of planning benchmark problems that predated the creation of PDDL, some planners admit "implicit" object declaration, meaning that object names can be used directly in action schemas without a prior declaration. In fact, since predicates, functions, and objects are all defined by the way they appear in action schemas and in the initial state and goal sections of the problem, one may take the view that all declarations are unnecessary and permit all symbols except actions to be declared implicitly. The advantage of adhering to PDDL's explicit declarations is that it makes possible additional validation of the domain, such as checking consistency between the declaration of, for example, a predicate and its use in action schemas. This can help detect modelling errors that would otherwise manifest in ways that make them much harder to debug.

2.4 ADVANCED MODELLING EXAMPLES

In this section we will present a number of slightly more complex examples of problems and how they can be modelled in the subset of PDDL that we have introduced so far. While the examples in Section 2.1 served to introduce the language, the purpose of the following examples is to demonstrate its potential, and in particular some common modelling "tricks" that may not be obvious.

The first example, in Section 2.4.1, shows how we can model a complex action as a sequence of simpler steps, and define actions for each of the steps such that they must always occur in a plan as sequence that makes up an instance of the original action. The second example, in Section 2.4.2, shows we can model a complex goal condition by separating the plan into two distinct phases and verifying the goal condition one simple part at a time.

2.4.1 SORTING BY REVERSALS

Suppose we are given a cyclic permutation of the numbers $1, \ldots, n$, and we wish to sort it into the identity permutation, so that 1 is followed by 2, which is followed by 3, etc., up to n which is followed by 1. The operation we can apply to change the permutation is to take a segment, of arbitrary length, and reverse it. It is always possible to sort the permutation using at most $n - 1$ reversals, but what is the smallest number of reversals? Figure 2.3 shows an example of a permutation of size 5 sorted by two reversals.

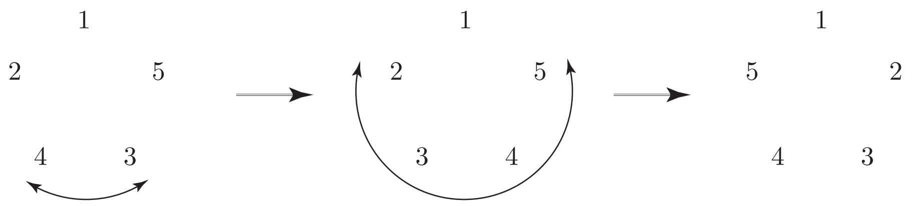

Figure 2.3: Example of sorting a cyclic permutation by reversals.

This problem is related to the problem of comparing genomes [Hannenhalli and Pevzner, 1995]. A genome is an ordered collection of genes. Simple genomes, like those of bacteria or mitochondria, take the form of single cyclic permutation. While insertion, deletion, and substitution of individual nucleotides mutates the genes, changes to their arrangement in the genome at evolutionary scale appear as reversals and transpositions. Because DNA is directional, genomes are usually represented as signed permutations. Here, however, we will only describe a way to model sorting of an unsigned cyclic permutation by reversals. A complete PDDL model of the rearrangement of signed genomes by reversals and transpositions is described by Haslum [2011], and can be found at `editor.planning.domains/pddl-book/ged`.

First, let's consider how to represent states. Given *n* objects, and we can represent a cyclic permutation with a single predicate `(next ?x ?y)`, meaning that `?y` is the next element after `?x`. The direction in which we enumerate the elements of the cycle does not matter; we will assume it is done clockwise. Thus, the leftmost permutation in Figure 2.3 is represented by the facts

```
(next n1 n5) (next n5 n3) (next n3 n4) (next n4 n2) (next n2 n1)
```

For this to represent a cyclic permutation, for every element *i* there must be exactly one element *j* such that (`next` *i* *j*) is true. This property is another example of an intended *state invariant* of the problem, and we must define the actions of the domain so that they preserve it.

Defining an action that reverses a segment of any fixed length *k* is simple. For example, the following action reverses a segment of length 2, that is, two adjacent elements:

```
(:action reverse-2
 :parameters (?before_x ?x ?y ?after_y)
 :precondition (and (next ?before_x ?x)
                    (next ?x ?y)
                    (next ?y ?after_y))
 :effect (and (not (next ?before_x ?x))
              (not (next ?x ?y))
              (not (next ?y ?after_y))
              (next ?before_x ?y)
              (next ?y ?x)
              (next ?x ?after_y))
 )
```

Note that as long as there are at least 3 elements, the invariant property that the state is a cyclic permutation guarantees that `?x`, `?y`, and `?before_x`/`?after_y` are distinct (though if there are only three elements, `?before_x` and `?after_y` will be the same). Thus, the action deletes the fact (`next ?x` *i*), which must be true when the action is applied, and adds a different fact (`next ?x` *j*), and the same for `?y` and `?before_x`, so it preserves the invariant property.

What makes the general problem more difficult to model is that the size of the segment to be reversed can be arbitrarily large. It is of course bounded by the number of elements in the problem, but if we were to write a different action for each size, we would need a different domain definition for each size of problem. These actions would also have a large number of parameters, which can degrade the efficiency of planners (we discuss this issue further in Section 2.5.6).

Our solution to this problem is to break the action of reversing a segment into a sequence of steps, such that each step affects only a fixed number of elements, and defining an action for each step. We can think of these steps as implementing an "algorithm" for reversing a segment of any length. The algorithm that we will use is as follows.

1. Select two elements *first* and *last*, marking the beginning and end of the segment to be reversed.
2. While *first* is not equal to *last*,
   - 2(a) Let *before-first* and *after-first* be the elements before and after first, respectively, in the current permutation.
   - 2(b) Remove element *first* from its current position and insert it after *last*.
   - 2(c) Change *first* to refer to *after-first*, and repeat step 2.

The steps of the algorithm are illustrated in Figure 2.4 with an example of reversing the segment 5–4–3–2.

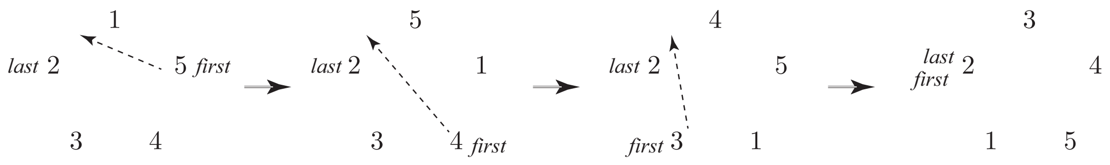

Figure 2.4: Illustration of the step-by-step process for reversing a segment of arbitrary length.

In the PDDL model, each step of the algorithm must be taken by an action, and we have to define the actions such that in any valid plan, they can only appear in the correct order and as a complete instance of a reversal. This means once a reversal has begun, it must be completed before a new reversal is started. To ensure this, we will use a predicate `(idle)` that is true exactly when there is no ongoing reversal. Two other predicates, `(is_first ?x)` and `(is_last ?x)`, will indicate that a reversal is in progress and which elements are the current *first* and *last*. The complete domain definition is as follows:

```
(define (domain sorting_by_reversal)
  (:requirements :strips :equality)

  (:predicates
   (next ?x ?y)   ; ?y is after ?x in clockwise order
   (idle)         ; no ongoing reversal operation
   (is_first ?x) ; the "first" element of the current reversal
   (is_last ?x)) ; the "last" element of the current reversal

  ;; Begin a new reversal of the segment between ?first and ?last:
  (:action begin_reversal
   :parameters (?first ?last)
   :precondition (idle)
   :effect (and (not (idle))
                (is_first ?first)
                (is_last ?last))
   )

  ;; Move one element, and update is_first:
  (:action move_element
   :parameters (?before_first ?first ?after_first ?last ?after_last)
   :precondition (and (is_first ?first)
                      (is_last ?last)
                      (not (= ?first ?last))
                      (next ?before_first ?first)
                      (next ?first ?after_first)
                      (next ?last ?after_last))
   :effect (and (not (next ?before_first ?first))
                (not (next ?first ?after_first))
                (next ?before_first ?after_first)
                (not (next ?last ?after_last))
                (next ?last ?first)
                (next ?first ?after_last)
                (not (is_first ?first)) ; update is_first
                (is_first ?after_first))
   )

  ;; End the reversal when "first" and "last" are the same:
```

```
(:action end_reversal
 :parameters (?element)
 :precondition (and (is_first ?element)
                    (is_last ?element))
 :effect (and (not (is_first ?element))
              (not (is_last ?element))
              (idle))
 )

)
```

PDDL Example 8: The sorting-by-reversals domain.

Note that we need an inequality to ensure that the `move_element` action can not be applied past the end of the reversed segment.

Because each segment reversal in this model consists of a number of actions, which varies with the length of the segment, the length of the plan does not equal the number of reversals. To actually solve the problem of finding the minimum number of reversals needed needed to sort a given permutation, we need to introduce action costs, such that the cost of beginning a new reversal outweighs the cost of the other actions sufficiently to ensure that the plan with the fewest reversals has the smallest cost. One way to achieve this is to assign a cost of 1 to `begin_reversal` and 0 to the other actions.

2.4.2 DEADLOCK DETECTION

A *deadlock* is a situation in which the execution of a collection of concurrent processes cannot progress because they are all waiting for one another. This kind of situation can arise in concurrent computer systems if they are not designed to avoid such behaviour. If the concurrent processes are modelled as a collection of finite state machines then the question of whether the system can reach a deadlocked state is an example of a model checking problem [Clarke et al., 1993]. As mentioned in Section 1.4.2, planning and model checking are closely related, and here we will show how deadlock detection can also be modelled as a planning problem. The problem that we formulate is to find a plan for the system to reach a deadlocked state. To prove that a system is free of (reachable) deadlocks, we thus need to use a complete planner that is able to prove no plan exists for the problem instance.

To illustrate, we will use the "dining philosophers" problem [see, for example, Hoare, 1985]. This is an artificial example, often used to explain the concept of a deadlock because of its simplicity. *n* philosophers are seated for dinner at a round table, and between each pair of philosophers is one fork. The philosophers' default state is to be thinking, but at any time any one of them can decide to eat. He will then pick up the two forks to either side, eat for a while and then return the forks, making them available for his neighbours to use. Importantly, each philosopher picks up the fork to his right first, and then the fork to his left. A deadlock can occur because once a philosopher has picked up the right fork, he is committed to picking up both forks, and will not return the one he is holding until he has done so. The deadlock occurs if all philosophers pick up their right-hand fork, since they are then all waiting for their left neighbour to return a fork, but none is able to do so. Figure 2.5 shows a philosopher and the adjacent forks as a set of partially synchronised automata. The forks in the example represent resources with mutually exclusive access. In a concurrent computer system, these could be file locks, network access tokens, or some other resource.

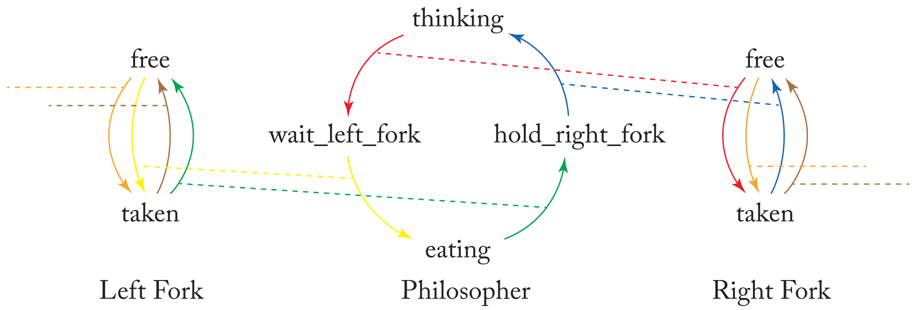

Figure 2.5: Finite automata representing a dining philosopher and the two adjacent forks. Synchronised transitions are linked with dashed line, and coloured the same. Transitions of the two forks are analogously synchronised with the neighbouring philosophers to the left and right.

Modelling the transitions of the automata as action schemas in PDDL is straightforward. The difficulty of this problem is how to model the goal. A deadlock state is one in which no action is applicable. Thus, the goal condition can be expressed as the conjunction over all actions of the negation of their preconditions. This, however, does not fall within the restricted form that goal formulas must have in the STRIPS subset of PDDL.

The trick that we will use to model the problem is similar to that of the previous example, formulating a set of actions that each verifies a simple part of the goal condition. For this, we will need to split the plan into two distinct *phases*: actions in the first phase bring the system to a deadlocked state, while actions in the second phase verify the goal condition. Once the goal-checking phase has begun, no actions that change the state can be taken. This ensures that all parts of the goal are verified in the same state. The separation of the two phases is controlled by a predicate, say `(normal)`, which is set to true in the initial state, and is a precondition of all normal (state-changing) actions. The goal-verifying actions delete this predicate, and after it has been deleted no action can make it true again. This ensures that once the goal-checking phase of the plan has begun, by taking any goal-verifying action, it cannot return to the state-changing phase. The complete domain definitions is as follows:

```
(define (domain philosophers)
  (:requirements :strips :typing)

  (:types philosopher fork)

  (:predicates
   ;; states of a philosopher:
   (thinking ?p - philosopher)
   (wait_left_fork ?p - philosopher)
   (eating ?p - philosopher)
   (hold_right_fork ?p - philosopher)
   ;; states of a fork:
   (free ?f - fork)
   (taken ?f - fork)
   ;; describing the topology:
   (is_right_fork ?p - philosopher ?f - fork)
   (is_left_fork ?p - philosopher ?f - fork)
   ;; predicate to express the goal:
   (deadlocked ?p - philosopher)
   ;; the phase control:
   (normal))

  ;; Actions for the normal (state-changing) phase:

  (:action pick_up_right
   :parameters (?p - philosopher ?f - fork)
   :precondition (and (is_right_fork ?p ?f)
                      (thinking ?p)
                      (free ?f)
                      (normal))
   :effect (and (not (thinking ?p))
                (wait_left_fork ?p)
                (not (free ?f))
                (taken ?f))
   )

  (:action pick_up_left
   :parameters (?p - philosopher ?f - fork)
   :precondition (and (is_left_fork ?p ?f)
                      (wait_left_fork ?p)
                      (free ?f)
                      (normal))
   :effect (and (not (wait_left_fork ?p))
                (eating ?p)
                (not (free ?f))
                (taken ?f))
   )

  (:action put_down_left
   :parameters (?p - philosopher ?f - fork)
   :precondition (and (is_left_fork ?p ?f)
                      (eating ?p)
                      (taken ?f)
                      (normal))
   :effect (and (not (eating ?p))
                (hold_right_fork ?p)
                (not (taken ?f))
                (free ?f))
   )

  (:action put_down_right
   :parameters (?p - philosopher ?f - fork)
   :precondition (and (is_right_fork ?p ?f)
                      (hold_right_fork ?p)
                      (taken ?f)
                      (normal))
   :effect (and (not (hold_right_fork ?p))
                (thinking ?p)
                (not (taken ?f))
                (free ?f))
   )

  ;; Actions for the goal-checking phase:

  (:action deadlock_right
   :parameters (?p - philosopher ?f - fork)
   :precondition (and (is_right_fork ?p ?f)
                      (thinking ?p)
```

```
                       (taken ?f)
                       (normal))
   :effect (and (not (normal))
                (deadlocked ?p))
   )

  (:action deadlock_left
   :parameters (?p - philosopher ?f - fork)
   :precondition (and (is_left_fork ?p ?f)
                       (wait_left_fork ?p)
                       (taken ?f)
                       (normal))
   :effect (and (not (normal))
                (deadlocked ?p))
   )

  )
```

The four possible states of a philosopher are represented by predicates `(thinking ?p)`, `(wait_left_fork ?p)`, `(eating ?p)`, and `(hold_right_fork ?p)`. Again, we have a state invariant: for every philosopher, exactly one of these is true in every reachable state.

The two actions for verifying that a philosopher is deadlocked correspond to the `thinking` and `wait_left_fork` states, since in the other two states he is holding both or one forks and nothing can prevent him from releasing them. Both of them add a fact `(deadlocked ?p)`, which is used to express the goal that all philosophers are in a deadlocked state. The following is a small example problem with three philosophers:

```
(define (problem three)
  (:domain philosophers)

  (:objects Kant Heidegger Wittgenstein - philosopher
            f1 f2 f3 - fork)

  (:init
   (is_right_fork Kant f1)
   (is_right_fork Heidegger f2)
   (is_right_fork Wittgenstein f3)
   (is_left_fork Kant f3)
   (is_left_fork Heidegger f1)
   (is_left_fork Wittgenstein f2)
   (thinking Kant)
```

```
    (thinking Heidegger)
    (thinking Wittgenstein)
    (free f1)
    (free f2)
    (free f3)
    (normal))

  (:goal (and (deadlocked Kant)
              (deadlocked Heidegger)
              (deadlocked Wittgenstein)))
  )
```

A possible valid plan for this problem is as follows:

```
(pick_up_right Kant f1)
(pick_up_right Heidegger f2)
(pick_up_right Wittgenstein f3)
(deadlock_left Kant f3)
(deadlock_left Heidegger f1)
(deadlock_left Wittgenstein f2)
```

2.5 EXPRESSIVENESS AND COMPLEXITY

Since PDDL is a language for expressing a certain kind of computational problem, the expressivity of the language is closely related to the computational complexity of solving those problems. In this section, we will review a few classical and well-known complexity results for the class of problems expressible in two subsets of the STRIPS fragment PDDL. The two subsets are without and with parameterised predicates and actions. We refer to the subset of the language not using parameters as *ground STRIPS*. As we will show below, using parameters allows for an exponentially more compact encoding of certain problems. Because the computational complexity of a problem is characterised as a function of the size of the problem instance, the introduction of parameters translates into an exponential increase in the worst-case complexity of the general planning problem.

We assume that the reader is familiar with standard computational complexity notions, such as reducibility and complexity classes. For an introduction, see Garey and Johnson [1979]. The complexity classes we will deal with here are only PSPACE and EXPSPACE.

Let D be a domain definition and P a problem definition for D. We use $|\cdot|$ to denote the size of an object (such as D or P). We distinguish the following computational problems.

- Plan validation: Given D, P, and a plan π, decide whether π is valid for D and P.
- Plan existence: Given D and P, decide whether there exists a plan that is valid for D and P.

- Plan generation: Given D and P, output a plan that is valid for D and P, or determine that no such plan exists.
- Optimal plan cost (generation): Given D and P, output the minimum cost of any plan (resp. a minimum-cost plan) for D and P, or determine that no such plan exists.

These problems are clearly related. For example, if we can solve (optimal) plan generation, then we can also answer plan existence at no more than the same computational cost.

2.5.1 PLAN VALIDATION

The first observation is that plan validation is polynomial in $|D|$, $|P|$, and $|\pi|$. To validate the plan, we need to verify that each ground action name in π is a valid instance of an action schema in D, construct each state in the induced state sequence, and test the applicability of each action in its respective state. To do this, we only need to keep track of the "current" state, meaning the one reached after executing the first k actions in the plan. Once the whole plan has been executed, we can test whether the final state satisfies the goal formula.

Finding the action schema in D that a ground action name corresponds to and checking that it has the right number of arguments for the parameters of the schema can be done in time linear in $|D|$. To check that the instantiation is type-correct, we need to compute the IS-A relation, which requires at worst time cubic in $|\text{types}(D)|$ (to compute the transitive closure) and linear in $|\text{objects}(P)|$. Given a state s, evaluating a ground formula φ in s can be done in time linear in $|\varphi| \times |s|$ (using time $|s|$ to decide the truth of each ground fact in φ). It remains only to show that the size of the current state will remain bounded. The initial state is part of the problem definition, and therefore bounded by $|P|$. The number of facts added by each ground action in the plan must be less than $|D|$. Thus, the size of each state in the induced state sequence is bounded by $|D| \times |\pi|$.

2.5.2 BOUNDS ON PLAN LENGTH: GROUND STRIPS

In a domain and problem without parameters, each predicate is a single ground fact. Thus, the size of the set of possible ground facts is at most $m = |\text{predicates}(D)|$, which means the number of distinct possible states is at most 2^m.

If there is a valid plan for D and P, then there is also a valid plan of length at most $2^m - 1$. Suppose that $\pi = n_1, \ldots, n_l$ is a valid plan such that $l > 2^m - 1$. The length of the induced state sequence is then greater than 2^m, which means it must contain at least one repeated state. Suppose that states i and j are the same: then $\pi' = n_1, \ldots, n_i, n_{j+1}, \ldots, n_l$ is also a valid plan for D and P. Because the state s_i reached by applying $n_1, \ldots, n_i$ is the same as the state s_j reached by applying $n_1, \ldots, n_j$, action n_{j+1} is applicable also in s_i, and applying it to s_i results in a state that is again the same as the state that results from applying it to s_j. Thus, the whole suffix $n_{j+1}, \ldots, n_l$ can be applied from state s_i, and results in the same final state as the origial plan π. Since π is valid, the goal formula holds in this state. The length of π' is strictly less than that of π. Thus, π can be replaced with successively shorter valid plans, until it is not longer than $2^m - 1$.

Next, we will show that $2^m - 1$ is also a lower bound on plan length, that is, that there exist ground STRIPS domains and problems which require exponential-length plans. The simplest example of such a domain and problem is one that models a binary counter. The domain has m predicates, `(bit_0)`, `(bit_1)`, ..., `(bit_`$(m-1)$`)`, and m actions, `(set_0)`, ..., `(set_`$(m-1)$`)`. Action `(set_`i`)` has the precondition `(and (bit_0) ... (bit_`$(i-1)$`))` and the effect `(and (bit_`i`) (not (bit_0)) ... (not (bit_`$(i-1)$`)))`. Action `(set_0)` has no precondition, and only the effect `(bit_0)`. In other words, the action that sets the ith bit to 1 (true) requires all lower-indexed bits to be set, and resets them to 0 (false). The initial state is empty (all facts are false) and the goal formula requires all m predicates to be true. We can show by induction that to reach a state in which `(bit_0)`, ..., `(bit_`i`)` are true requires $2^{i+1} - 1$ actions. As the base case, to make `(bit_0)` true requires only one action, and $2^1 - 1 = 1$. To make `(bit_0)`, ..., `(bit_`i`)` all true requires first making `(bit_0)`, ..., `(bit_`$(i-1)$ true, which, by inductive assumption, requires $2^i - 1$ actions. We can then apply action `(set_`i`)` which adds `(bit_`i`)`, but because this action deletes `(bit_0)`, ..., `(bit_`$(i-1)$, another $2^i - 1$ actions are needed to make those facts true again. The total plan length is $(2 \times 2^i - 1) + 1 = 2^{i+1} - 1$.

2.5.3 COMPUTATIONAL COMPLEXITY OF GROUND STRIPS PLANNING

The plan length bounds above have as an immediate consequence that the (optimal) plan generation problem requires, in the worst case, exponential time, simply because the plan that is generated may be exponentially long.

The plan validation procedure together with the observation that in ground STRIPS the size of the set of possible ground facts is linearly bounded by the domain size, given that the plan existence problem can be solved in nondeterministic polynomial space, by a guess-and-check algorithm. The algorithm maintains, using polynomial space, only the current state, and nondeterministically selects actions to apply until the state satisfies the goal formula. Since NPSPACE is equal to PSPACE [Savitch, 1970], this means the plan existence problem for ground STRIPS is in PSPACE.

This problem is also PSPACE-hard [Bylander, 1991]. A reduction can be made directly from the acceptance problem for a polynomially space-bounded Turing machine.

2.5.4 COMPUTATIONAL COMPLEXITY OF PARAMETERISED STRIPS PLANNING

PDDL with parameters allows certain problems to be expressed exponentially more compactly than in the corresponding ground language. If there are $n = |\text{objects}(P)|$ objects, a predicate with k parameters can potentially be instantiated into n^k ground facts. Hence, the number of possible ground facts is already exponential in $|D| + |P|$, and the number of possible states is doubly exponential in the size of the domain and problem. By the argument above, if there exists any plan for a given domain and problem there also exists a plan of length at most 2^{n^k}, where k is the maximum arity of predicates in the domain, and the plan existence question can be answered by a nondeterministic algorithm using single-exponential space.

To show a matching lower bound on plan length, we will construct a model of a binary counter with 2^k bits, which can count up to 2^{2^k}, and construct a problem whose plan requires incrementing the counter at least $2^{2^k}/2$ times. We will use a predicate with $k+1$ parameters, `(bit ?n`$_0$ `?n`$_1$ `...` `?n`$_{k-1}$ `?v)`, and just two objects, `zero` and `one`. The idea is that the first k parameters encode a number $i \in 0, \ldots, 2^k - 1$ in binary, and the last parameter is the value of the ith bit of the counter.

The action of incrementing this counter is complex, and we will need to encode it using a sequence of simpler steps. The procedure for incrementing the counter is as follows: let i denote an index, and $i_0, \ldots, i_{k-1}$ denote the binary representation of i. First, set $i = 0$. As long as the ith bit of the counter is 1 (i.e., `(bit` i_0 `...` i_{k-1} `one)` is true), reset that bit to 0 and increment i by one; when the ith bit is 0, set it to 1 and reset the index to $i = 0$. To encode this procedure, we will need predicates storing the binary representation of the index i. Because the index needs only k bits, we can do this with k predicates, as in the example of the ground binary counter above. However, because we will need to use the values of the index bits as arguments to the `bit` predicate we adopt a slightly different representation of this state variable: we use a unary predicate `(index_`j `?v)` for $j \in 0, \ldots, k-1$, and define the domain such that `(index_`j `zero)` is true in any state where $i_j = 0$ and `(index_`j `one)` is true in any state where $i_j = 1$. The actions that iterate through and reset counter bits are defined as follows:

```
(:action clear_counter_bit_and_inc_index_j
 :parameters (?n_(j+1) ... ?n_(k−1))
 :precondition (and (index_0 one)
                    ...
                    (index_(j−1) one)
                    (index_j zero)
                    (index_(j+1) ?n_(j+1))
                    ...
                    (index_(k−1) ?n_(k−1))
                    (bit one ... one zero ?n_(j+1) ... ?n_(k−1) one))
 :effect (and ((not (bit one ... one zero ?n_(j+1) ... ?n_(k−1) one))
               (bit one ... one zero ?n_(j+1) ... ?n_(k−1) zero)
               (not (index_0 one))
               (index_0 zero)
               ...
               (not (index_(j−1) one))
               (index_(j−1) zero)
```

```
                  (not (index_j zero))
                  (index_j one))
                 )
 )
```

There is one such action for each $j \in 0, \ldots, k-1$. The action that sets a counter bit is defined as follows:

```
(:action set_counter_bit_and_reset_index
 :parameters (?n_0 ... ?n_(k − 1))
 :precondition (and (index_0 ?n_0)
                     ...
                     (index_(k − 1) ?n_(k − 1))
                     (bit ?n_0 ... ?n_(k − 1) zero))
 :effect (and (not (bit ?n_0 ... ?n_(k − 1) zero))
              (bit ?n_0 ... ?n_(k − 1) one)
              (not (index_0 ?n_0))
              (index_0 zero)
              ...
              (not (index_(k − 1) ?n_(k − 1)))
              (index_(k − 1) zero))
 )
```

This action also resets the index to zero, so that the next iteration of the counter increment procedure can begin. Note that in this action's definition, we exploit that PDDL interprets an action effect that both adds and deletes a fact as making the fact true. This allows us to delete the currently true fact `(index_j ?n_j)` and add `(index_j zero)` without needing to create different cases for what the current value of each index bit is.

There is one more complication: to initialise the counter to zero, we need to set `(bit` i_0 `...` i_{k-1} `zero)` to true for every index $i \in 0, \ldots, 2^k - 1$. Writing these facts explicitly in the `:init` section would make the size of the problem definition exponential in k, and therefore invalidate the claim that the plan is doubly exponential in the size of the domain and problem. To overcome this issue, we can define a set of actions that iterate through the values of the index and add facts `(bit` i_0 `...` i_{k-1} `zero)` in the same way as we defined the actions above to clear the counter bits. To make sure that these actions are not interleaved with those that increment the counter, we again use a separation of the plan into two phases, like we did in the model of the deadlock detection problem in Section 2.4.2. Here, the two phases are the initialisation followed by a "normal" plan phase. We use two predicates, `(initialising)` and `(counting)`, to represent the current phase, and add `(initialising)` to the precondition of the actions that initialise the counter bits and, `(counting)` to the precondition of the actions that implement the counter increment procedure, making sure that each set of actions is applicable only in the correct phase. We also define an action `end_init_phase` that is applicable when the initialisation phase is complete, and which switches to the couting phase by deleting `(initialising)` and adding `(counting)`. In this way, the `:init` section of the problem must contain only the facts `(initialising)` and `(index_`j` zero)` for $j \in 0, \ldots, k-1$. The `:goal` of the problem is `(bit one ... one one)`, that is, for the highest bit of the counter to be set. To achieve this goal requires applying the action `set_counter_bit_and_reset_index` $2^{2^k}/2$ times.

A complete domain and problem definition for $k = 4$ can be found at `editor.planning.domains/pddl-book/counter4`. Note that although the problem only requires the counter to go up to 32,768, the step-wise increment actions and the initialisation phase mean that the shortest plan has 65,551 actions. This, however, does not mean that this plan is difficult to find, since in every reachable state of the problem there is only one applicable action. What the example demonstrates is just that it is possible for plans to be very long.

Using the same principle as in this example, a reduction like that which proves PSPACE-hardness of the ground STRIPS problem but exponentially more compact can be made through the use of parameters. In this way, it was shown that the plan existence question for the general STRIPS language is EXPSPACE-hard [Erol et al., 1991]. We will not repeat the details of the reduction here.

2.5.5 REVERSAL AND COMPLEMENT

The STRIPS fragment of PDDL has two intriguing closure properties: it is closed under *reversal*, meaning that for every STRIPS planning problem P there exists another problem P^R such that plans for P^R are exactly the plans for P in reverse, and it is closed under *complement*, meaning that for every STRIPS planning problem P there exists another problem $\overline{P}$ such that a plan exists for $\overline{P}$ if and only if no plan exists for P.

The reversibility of STRIPS planning was first demonstrated by Massey [1999]; a simplified construction of the reversed problem was proposed by Suda [2016]. We will not recount the details here.

The complementarity of plan existence for the STRIPS formalism is a consequence of the problem's computational complexity. Consider ground STRIPS: the plan existence problem is in PSPACE. PSPACE is closed under complement, which means the plan non-existence problem is in the same class, and thus there exists a polynomially space-bounded Turing machine that recognises precisely the set of unsolvable ground STRIPS planning problems. Since ground STRIPS planning is also PSPACE-hard, the acceptance problem of this Turing machine can be reduced to a ground STRIPS planning problem, via Bylander's [1991] reduction. The same argument can be applied to parameterised STRIPS which is EXPSPACE-complete, since EXPSPACE is also closed under complement.

2.5.6 PRACTICAL CONSIDERATIONS

The complexity results described above concern the worst case, in the sense of the hardest problems that can be expressed in the grounded and parameterised STRIPS subset of PDDL. When modelling a particular problem with the aim of solving it with a planner, however, the more relevant question is how to formulate the problem in a way that is most favourable to the planner, or at least does not unnecessarily disadvantage it. Although planners are intended to be domain-independent problem solvers, it must be recognised that the formulation of the domain can have a significant impact on the efficiency of planners. In other words, the question is how to write *efficient* PDDL.

There are no simple rules for how the domain formulation affects the efficiency of a planner, apart from the rule that *it depends on the planner*. Different PDDL planners have been implemented using a wide range of problem-solving techniques, which are affected in different ways. Yet, there are a few principles that are fairly generally applicable, which we will briefly discuss in the rest of this section.

Grounding

Although all PDDL planners accept parameterised domains as input, the majority internally convert this into a parameter-free, or grounded, representation, by instantiating actions and predicates with all valid combinations of arguments. This is done because the grounded representation lends itself to simpler and more efficient implementation of many solving techniques. However, since the number of ground instances of an action can be exponential in the number of parameters it has, this is a potentially costly operation. (Younes and Simmons [2002] compare the advantage and disadvantage of grounding in the context of a particular planning algorithm.)

As a first illustration, let us consider the different models of the Knight's Tour from Section 2.1.2. As we noted, there are 336 valid knights moves on a regular 8×8 chess board, so if we write a ground PDDL model it will have 336 actions. Our first parameterised domain formulation (Example 4) has a single action schema with two parameters. The problem has 64 objects, one for each square on the board, so there are $64^2 = 4096$ possible ground instances of this action schema. However, recall the static predicate `valid_move`, which is true only of those pairs of squares, listed in the initial state of the problem definition, which are the start and end of a valid Knight's move, and which is a precondition of the `move` action schema. Thus, 3760 of the `move` action instances have a precondition that is statically false. It is not difficult to filter out from the ground instances those with statically false preconditions, and one can safely assume that any good planner implementation does this in an efficient manner. Thus, we can reasonably say that this domain formulation is no more difficult than the ground version. Our second domain formulation (shown in Example 6) has two action schemas with four parameters, but only eight objects, leading to potentially $2 \times 8^4 = 8192$ instances, but again the static predicates in the preconditions of those action schemas limit the ground actions whose preconditions are not statically false to the same number.

As a second example, consider the problem of sorting by reversals presented in Section 2.4.1. The action schema for reversing a segment of fixed length two has four parameters, leading to n^4 potential instances on a permutation of size n. Unlike the previous example, this action schema has no static preconditions. The preconditions that force the four arguments to be a contiguous subsequence of the permutation mean that no more than n ground actions are applicable in any given state, but for every one of the n^4 possible instantiations there is some reachable state in which that ground action is applicable. Thus, a planner that grounds the problem is forced to generate all of them. If we write actions in the same way for reversing each fixed segment length up to $n-1$, we end up with more than n^{n-1} ground action instances, which will quickly become infeasible. The step-by-step domain formulation, shown in Example 8, in contrast, has a set of action schemas, with a fixed number of parameters, for any problem size, and thus no more than $n^2 + n^5 + n$ ground instances. (For large values of n, even this may be too much for a planner to handle. It can be reduced by further splitting up the `move_element` action schema into two steps that remove and insert the element, respectively.)

Symmetries in the Space of Plans

It is often the case that there are many different valid plans, even many different optimal plans, for a given planning problem. This can be the result of reordering of independent actions in the plan, of substituting different but equivalent objects, or arise simply because the problem can be solved in different ways. To illustrate, consider the logistics domain model introduced in Section 2.4, and suppose we have a problem instance with m identical trucks, and a valid plan in which only $k < m$ trucks are used; each truck takes l actions. There are $\binom{m}{k}$ different choices of which k trucks to use; for each choice of k trucks, there are $k!$ different assignments of the chosen trucks to the planned routes; and finally, for each assignment of trucks to routes, there are $\frac{(kl)!}{(l!)^k}$ different interleavings of their actions into a sequential plan. All these plans are equivalent, having the same length and cost.

In some cases, it is possible to rewrite the domain model to eliminate some of the symmetry. In the example of the logistics domain, we can define an ordering of the trucks and enforce that `drive` actions in the sequential plan occur in that order, i.e., that all `drive` actions involving the first truck must take place before any `drive` action involving the second truck, and so on. Since this serves only to prune different interleavings of independent actions, the ordering of the trucks can be arbitrary. We can also use the ordering to rewrite the domain model so that the possible choice of different subsets of identical trucks that are used in the plan and of the permutation of identical trucks between routes are eliminated.

However, unlike the potential combinatorial explosion due to grounding, which affects the majority of planners, the multiplicity of possible plans presents a problem only for planners based on certain problem solving paradigms, particularly cost-optimising planners that use explicit state-space search [e.g., Helmert and Röger, 2008]. Planners based on, for example, SAT encoding [e.g., Rintanen, 2010] or symbolic state-space search [e.g., Edelkamp et al., 2015], on the other hand may be able to exploit symmetries to compact the encoding of plans or states, so that a domain formulation that breaks the symmetries could potentially decrease their efficiency.