Numbers: PDDL 2.1

Planning and PDDL, Part III (Chapters 8–9)

Francisco Martín Rico

Intelligent Robotics Lab, URJC



Outline

1. Numeric fluents
2. Plan quality: metrics and action costs
3. Summary

1 Numeric fluents

2 Plan quality: metrics and action costs

3 Summary

What a fact cannot say

- A fact is true or false: right for a few possible values
- A battery is a **quantity**: a number on which the world does arithmetic
- 12 m at 1.5 units per meter take 18 units away, whatever the charge was
- With facts: one fact per charge, a table of subtractions by hand
- A number is a quantity when the planner must add, subtract or compare it



Resource consumed and replenished
battery, fuel, money, water



Capacity shared by several things
5 kg: 2 + 2 fit, a meal of 4 alone



Data that are numbers
never change, actions need them

Functions: numbers in the state

```
(:functions
  (battery ?r - robot)              ; charge left, 0 to 100
  (battery-capacity ?r - robot)     ; the most it can hold
  (consumption ?r - robot)          ; charge used per meter
  (distance ?from ?to - location)) ; meters between rooms
```

```
(= (battery r1) 100)
(= (battery-capacity r1) 100)
(= (consumption r1) 1)
```

- A **function** applied to objects is a **numeric fluent**: a number in the state
- The problem gives initial values in `:init`, with =
- No units, no range: keeping 0–100 is the job of the actions

a state: facts . . .

```
(robot-at r1 dock)
(free r1)
(item-at meds1 pharmacy)
```

```
(battery r1) = 100
```

. . . and values

- **Static**: no action changes it (distance, capacity); data
- **Dynamic**: some action changes it (charge); state
- Requirement `:numeric-fluents` (`:fluents` in PDDL 2.1)

Comparing and changing numbers

Numeric condition: < <= = >= >, operator first

```
(>= (battery ?r)
    (* (distance ?from ?to) (consumption ?r))))
```

- Expressions: numbers, fluents, + - * / in prefix form
- Anywhere a fact can go, goals included
- No "different from": `(not (= (load ?r) 0))`

Numeric effect: changes one fluent

- `(increase f e)` adds e
- `(decrease f e)` subtracts it
- `(assign f e)` replaces the value

Read the state before the action

All effects happen at once; every expression reads the values before the action. The order in which effects are written means nothing.

The battery in the model card

Step M: what the battery changes

Story. Moving uses charge in proportion to the meters. The dock recharges, and a recharge fills the battery.
Scope. In: the charge, the length of each connection, the charger. Out: how long a recharge takes, charge used by pick and drop, wear.
Assumptions. Same consumption per meter whatever the load. A robot never starts a move that it cannot finish.

- A robot stranded between rooms is not a state of the model
- So the model must forbid the move that would strand it
- That is a **precondition on a number**

The courier with a battery

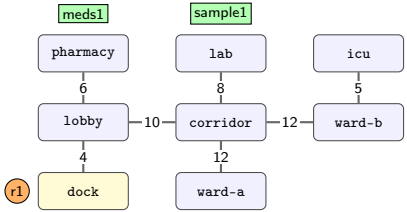
```
(:functions
  (battery ?r - robot)              ; charge left, 0 to 100
  (battery-capacity ?r - robot)     ; the most it can hold
  (consumption ?r - robot)          ; charge used per meter
  (distance ?from ?to - location)) ; meters between rooms

(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and
    (robot-at ?r ?from) (connected ?from ?to)
    ; enough charge for the whole stretch
    (>= (battery ?r)
        (* (distance ?from ?to) (consumption ?r))))
  :effect (and
    (not (robot-at ?r ?from)) (robot-at ?r ?to)
    (decrease (battery ?r)
              (* (distance ?from ?to) (consumption ?r)))))
```

```
(:action charge
  :parameters (?r - robot ?l - location)
  :precondition (and
    (robot-at ?r ?l) (has-charger ?l)
    (< (battery ?r) (battery-capacity ?r)))
  :effect (and
    (assign (battery ?r) (battery-capacity ?r))))
```

- Precondition and effect of `move` use the **same expression**; nothing in PDDL ties them
- `charge` assigns the capacity
- `(< (battery ?r) ...)`: no charging a full battery, no useless states
- `pick`, `drop`: the typed courier

Running it: one delivery



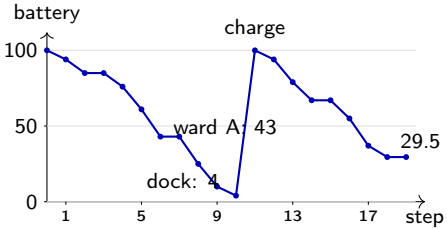
- $4 + 6 + 6 + 10 + 12 = 38$ m, charge for 100
- Step E, trivial test: the numbers broke nothing

```
$ kairos check courier-battery-domain
    .pddl courier-battery-p01.pddl
valid, plannable
  typing: full
  numeric-fluents: full
$ kairos courier-battery-domain.pddl
    courier-battery-p01.pddl
; cost = 7
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
...
(drop r1 meds1 ward-a)
```

Two deliveries: the planner recharges

```
$ kairos courier-battery-domain.pddl
     courier-battery-p02.pddl
; cost = 19
(move r1 dock lobby)
...
(drop r1 meds1 ward-a)
(move r1 ward-a corridor)
(move r1 corridor lobby)
(move r1 lobby dock)
(charge r1 dock)
(move r1 dock lobby)
...
(drop r1 sample1 icu)
```

Consumption 1.5, two deliveries.



- Nobody asked for it: without it a later move is not applicable
- Ward A with 43; the second delivery needs 67.5

Following the numbers

`--dump=plan-trace` writes the state after every step (only what can change):

```
$ kairos courier-battery-domain.pddl courier-battery-p02.pddl \
    --no-anytime --dump=plan-trace --dump-dir trace > plan.txt
$ grep -A5 "step 1[01]:" trace/plan-trace.txt
; step 10: (move r1 lobby dock)
  ...
  (= (battery r1) 4.000000)
; step 11: (charge r1 dock)
  ...
  (= (battery r1) 100.000000)
```

Exercising the model: the test set

Step E: the test set of the battery

Trivial	p01: one delivery, charge to spare. Plan of 7 actions, no charge.
Typical	p02: two deliveries. Plan of 19 actions, one visit to the dock where the trace says it must be.
No plan	p03: the robot in the ICU with 20 units, an item for the lab 25 m away. There must be no plan.
Cheat	p04: the goal is a battery above its capacity, `(> (battery r1) 100)`. There must be no plan.

The validator does the arithmetic too. p02 without its charge:

```
$ kairos validate courier-battery-domain.pddl \
    courier-battery-p02.pddl courier-battery-p02-nocharge.plan
courier-battery-p02-nocharge.plan:12: error: step 11 (move r1 dock lobby) at
    10.000: precondition false: (>= (battery ?r) (* (distance ?from ?to) (
    consumption ?r))): 4 >= 6 is false [validate.precondition_failed]
 12 | (move r1 dock lobby)
plan invalid
```

The two ends of a quantity

```
$ kairos courier-battery-domain.pddl courier-battery-p03.pddl
note: no plan: every state reachable from the initial one was searched (22
    states) and none satisfies the goal. [orchestrator.why]
kairos: unsolvable (exhausted)
```

A first draft: the charger added a fixed amount

```
:precondition (and (robot-at ?r ?l) (has-charger ?l))
:effect (and (increase (battery ?r) 50)))
```

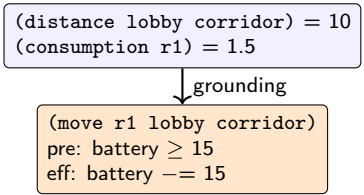
```
$ kairos courier-battery-naive-domain.pddl \
    courier-battery-naive-p04.pddl
; cost = 1
(charge r1 dock)
```

- Cheat test: 110 units in a battery of 100
- With `assign`: no plan
- **Numbers have no bounds of their own**

Numbers that never change

```
$ kairos courier-battery-domain.pddl courier-battery-p02.pddl -v
...
[   0.001s] [ground] grounded in 0.1 ms: 27 atoms, 1 numeric fluents, 47
    actions, 0 durative actions, 0 axioms (97 Datalog facts, 9 rules, 0
    instances pruned)
```

- 17 numbers in `:init`, **1 numeric fluent**
- Static functions are folded into the ground actions, as static facts were
- Data should be data: one `move`, the map in the problem



A load capacity

```
(:functions
  (load ?r - robot) - number       ; kg it carries now
  (capacity ?r - robot) - number   ; kg it can carry
  (weight ?i - item) - number      ; kg of one item
(:action pick
  :parameters (?r - robot ?i - item ?l - location)
  :precondition (and
    (robot-at ?r ?l) (item-at ?i ?l)
    ; the item must still fit
    (<= (+ (load ?r) (weight ?i)) (capacity ?r)))
  :effect (and
    (not (item-at ?i ?l)) (carrying ?r ?i)
    (increase (load ?r) (weight ?i))))

(:action drop
  :parameters (?r - robot ?i - item ?l - location)
  :precondition (and (robot-at ?r ?l) (carrying ?r ?i))
  :effect (and
    (not (carrying ?r ?i)) (item-at ?i ?l)
    (decrease (load ?r) (weight ?i)))))
```

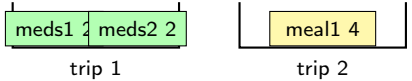
- `free` is gone: a comparison takes its role
- `weight`, `capacity`: static; only `load` changes
- `(load ?r)` **summarizes** the weights carried (pattern of chapter 7)

The duty of a summary

Every action that changes what is summarized updates the summary. An `increase` without the `decrease`: a tray that fills up for ever.

Three items, 8 kg, a robot for 5

```
    (robot-at r1 lobby)
    (= (load r1) 0) (= (capacity r1) 5)
    (item-at meds1 lobby) (= (weight meds1) 2)
    (item-at meds2 lobby) (= (weight meds2) 2)
    (item-at meal1 lobby) (= (weight meal1) 4))
  (:goal (and (item-at meds1 ward-a) (item-at meds2 ward-a)
              (item-at meal1 ward-a))))
```



- The only grouping that makes two trips enough (with `free`: three)
- Meal of 6 kg (p02): unsolvable, 1296 states

```
$ kairos courier-load-domain.pddl
     courier-load-p01.pddl
; cost = 12
(pick r1 meds2 lobby)
(pick r1 meds1 lobby)
...
(drop r1 meds2 ward-a)
(drop r1 meds1 ward-a)
...
(pick r1 meal1 lobby)
...
(drop r1 meal1 ward-a)
```

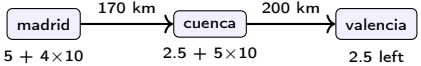
A truck and its fuel

```
(:action drive
  :parameters (?t - truck ?from ?to - city)
  :precondition (and
    (truck-at ?t ?from) (road ?from ?to)
    (>= (fuel ?t) (/ (distance ?from ?to) (km-per-liter ?t))))
  :effect (and
    (not (truck-at ?t ?from)) (truck-at ?t ?to)
    (decrease (fuel ?t)
              (/ (distance ?from ?to) (km-per-liter ?t)))))

(:action refuel
  :parameters (?t - truck ?c - city)
  :precondition (and
    (truck-at ?t ?c) (has-station ?c)
    (<= (+ (fuel ?t) 10) (tank ?t)))   ; 10 more must fit
  :effect (increase (fuel ?t) 10)))
```

- Consumption: km over km per liter
- A capacity kept by a precondition: 10 more must fit

```
$ kairos truck-domain.pddl
    truck-p01.pddl
; cost = 11
(refuel t1 madrid)
...
(drive t1 madrid cuenca)
(refuel t1 cuenca)
...
(drive t1 cuenca valencia)
```



The water jugs

```
; everything in ?from fits in ?to
(:action pour-all
  :parameters (?from ?to - jug)
  :precondition (and
    (not (= ?from ?to))
    (> (amount ?from) 0)
    (<= (+ (amount ?from) (amount ?to)) (volume ?to)))
  :effect (and
    (increase (amount ?to) (amount ?from))
    (assign (amount ?from) 0)))

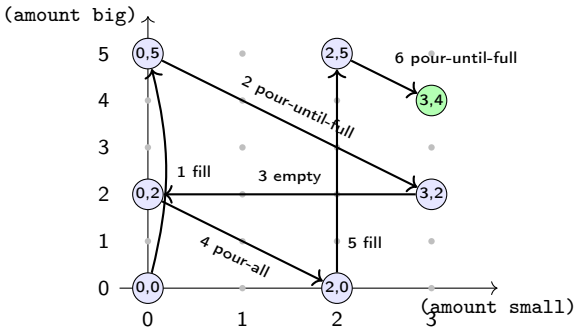
; ?to gets full and something stays in ?from
(:action pour-until-full
  :parameters (?from ?to - jug)
  :precondition (and
    (not (= ?from ?to))
    (< (amount ?to) (volume ?to))
    (> (+ (amount ?from) (amount ?to)) (volume ?to)))
  :effect (and
    (decrease (amount ?from) (- (volume ?to) (amount ?to)))
    (assign (amount ?to) (volume ?to)))))
```

- Jugs of 3 and 5 liters, no marks; measure exactly 4
- Pouring has two outcomes: **two actions**
- `pour-all`: the first effect reads `(amount ?from)`, the second sets it to 0
- Both read the old amount: swapping the lines changes nothing
- Goal: a numeric condition, `(= (amount big) 4)`

A plan as a path among pairs of numbers

```
$ kairos jugs-domain.pddl
    jugs-p01.pddl
; cost = 6
(fill big)
(pour-until-full big small)
(empty small)
(pour-all big small)
(fill big)
(pour-until-full big small)
```

Jugs of 4 and 6, goal 3 (p02): every amount is even; 20 states, unsolvable.



The state has no facts: it is a pair (small, big).

Scaling: a dilution

```
(:functions
  (concentration ?t - tube)    ; mg per ml
  (volume ?t - tube)           ; ml of liquid in the tube
  (size ?t - tube))            ; ml the tube holds

(:action dilute
  :parameters (?t - tube)
  :precondition (<= (* 2 (volume ?t)) (size ?t))
  :effect (and (scale-up (volume ?t) 2)
               (scale-down (concentration ?t) 2)))
(:action discard-half
  :parameters (?t - tube)
  :precondition (>= (volume ?t) 2)
  :effect (scale-down (volume ?t) 2)))
```

- `(scale-up f e)` multiplies, `(scale-down f e)` divides
- For proportions: interest, doubling, discounts

```
$ kairos dilution-domain.pddl
    dilution-p01.pddl
; cost = 4
(dilute t1)
(dilute t1)
(discard-half t1)
(dilute t1)
```



Goal: `(<= (concentration t1) 10)`, `(>= (volume t1) 4)`; tube of 10 ml.
An inequality accepts many states.

Counting: stock in a warehouse

```
(:functions
  (stock ?p - product)     ; units on the shelf
  (batch ?p - product)     ; units that one restocking brings
  (packed ?p - product)    ; units already in the box
  (box-space))             ; units that still fit in the box

(:action pack
  :parameters (?p - product)
  :precondition (and (>= (stock ?p) 1) (>= (box-space) 1))
  :effect (and (decrease (stock ?p) 1)
               (increase (packed ?p) 1)
               (decrease (box-space) 1)))

; the supplier only comes when the shelf is empty
(:action restock
  :parameters (?p - product)
  :precondition (= (stock ?p) 0)
  :effect (increase (stock ?p) (batch ?p))))
```

```
$ kairos warehouse-domain.
    pddl warehouse-p01.
    pddl
; cost = 6
(pack masks)
(pack masks)
(pack gloves)
(restock gloves)
(pack gloves)
(pack gloves)
```

- Counted, not named

Values that were never given

Omitted in `:init`: **undefined**, not zero

Actions that read it never apply, silently

Delete `(= (battery r1) 100)` from the first problem:

```
$ kairos check courier-battery-domain.pddl \
    courier-battery-undefined.pddl
valid, plannable
  typing: full
  numeric-fluents: full
$ kairos courier-battery-domain.pddl courier-battery-undefined.pddl
note: no plan: every state reachable from the initial one was searched (2
    states) and none satisfies the goal. [orchestrator.why]
kairos: unsolvable (exhausted)
```

A one-way door built by forgetting a line

Delete `(= (distance ward-a corridor) 12)`:

```
$ kairos courier-battery-domain.pddl
    courier-battery-oneway.pddl
; cost = 22
...
(drop r1 meds1 corridor)
...
(pick r1 meds1 corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

Medicines left in the corridor until the end.



no distance: never applies

Why not zero?

A zero distance makes a move free. An undefined value blocks actions: doing too little is easier to notice than doing too much.

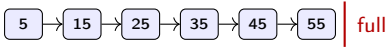
- Write values next to their facts
- A reachability test in step E

Why numbers make planning harder

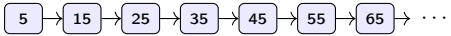
- Facts: finitely many states, so "unsolvable" can be proved
- A fluent may take infinitely many values
- Truck without the tank limit:

```
(:action refuel
  :parameters (?t - truck ?c - city)
  :precondition (and (truck-at ?t ?c) (has-station ?c))
  :effect (increase (fuel ?t) 10)))
```

With a tank of 60 liters



Without a limit



The battery, the jugs and the truck with its tank cannot grow: a search that remembers states comes to an end.

Bounded or endless

A road of 40 liters, 5 in the tank, goal: exactly 12. With the tank of 60:

```
$ kairos truck-domain.pddl truck-p03.pddl
note: no plan: every state reachable from the initial one was searched (16
    states) and none satisfies the goal. [orchestrator.why]
kairos: unsolvable (exhausted)
```

Without the limit, always one more state; no proof that more time will not help:

```
$ kairos truck-endless-domain.pddl truck-endless-p03.pddl \
    --timeout 5
note: no plan within the time limit of 5 s (1487655 states searched). More
    time may find one: --timeout (SolveOptions::timeout). [orchestrator.why]
kairos: timeout (timeout)
```

Not a weakness of one planner

Numeric planning is undecidable

For numeric fluents, Malte Helmert proved in 2002 that no algorithm can say in every case whether a plan exists. Two unbounded counters, add one, subtract one and a test for zero imitate any program: asking for a plan is asking whether a program stops.

- **Bounded** numbers: classical ground, "unsolvable" can be proved
- **Unbounded**: a planner may find a plan, but its silence proves nothing
- **More states**: a fluent multiplies them by its values
- **Weaker guidance**: the numeric relaxation ignores decreases; the courier never runs out in it

Bound every quantity that the world bounds; prefer few values to many.

The MODEL method: quantities

Step O: three questions for every property with a number in it

Is it a quantity? Does the model add, subtract or compare it? A room number or a priority among three is a name or a level.
Does only a threshold matter? If actions only ask "enough for one more move?", a fact (`charged`, `full`) may do. Keep the number when actions consume different amounts, or a goal or metric speaks of it.
Is it bounded? Smallest and largest value the world allows, and which precondition or `assign` keeps the fluent between them.

- Step D: does each action need or change each quantity; is every summary updated?
- Step E: two tests per quantity: more than there is, and a goal beyond a bound

The state table gains a kind of value

Property	About	Kind	Encoding and bounds
Charge	a robot	dynamic	`(battery ?r)`, 0 to the capacity; kept by the precondition of `move` and the `assign` of `charge`
Capacity of the battery	a robot	static	`(battery-capacity ?r)`
Consumption	a robot	static	`(consumption ?r)`, per meter
Length	two rooms	static	`(distance ?from ?to)`, both directions
Load	a robot	dynamic	`(load ?r)`, 0 to `(capacity ?r)`; kept by `pick`
Weight	an item	static	`(weight ?i)`

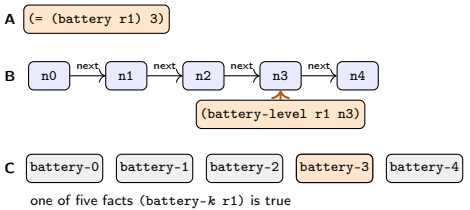
Every dynamic quantity names its bounds and the actions that keep them.

Alternatives: a battery counted in moves

Every move uses one unit, a full battery has four, pharmacy to ward A takes five moves.

- **A, a number**: `(>= (battery ?r) 1)`, `(decrease (battery ?r) 1)`
- **B, a chain of levels** (successor chain, chapter 7): `move` takes `?before ?after - level`
- **C, one predicate per level**: `battery-4` ... `battery-0`, stepped by `when`

Same plan of 10 actions in the three (B: `(move r1 dock lobby n4 n3)`).




```
(:action move
  :parameters (?r - robot ?from ?to - location
               ?before ?after - level)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to)
                     (battery-level ?r ?before)
                     (next ?after ?before))
  :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
               (not (battery-level ?r ?before))
               (battery-level ?r ?after)))
```

Three encodings compared

	A: number	B: chain	C: flags
Fidelity	Any amount: meters times consumption, fractions, different robots	Whole steps only; 15 units need 15 levels crossed, or one action per amount	As B, and the number of levels is fixed in the domain
Size	31 ground actions whatever the capacity; 1 numeric fluent	76 ground actions with 5 levels, 1516 with 101	31 ground actions, one conditional effect per level
Guidance	Finite only if bounded; numeric relaxation blind to running out	Finite; one state variable that classical heuristics handle well	As B
Clarity	Reads like the story	Two extra parameters per action, `next` facts in every problem	Long; a new capacity means editing the domain
Portability	Needs `:numeric-fluents`: not in classical planners	STRIPS and types (plus a negated equality in `charge`)	Conditional effects and negative preconditions

Verdict. A number when amounts are data (distances, weights, fractions, wide ranges). A chain for a small count that changes by one, above all with a classical planner. Flags for two or three named levels: a threshold, not a count.

Common mistakes (1)

Consumption without precondition

Too good: p03 (20 units, 25 m) has a plan.

```
$ kairos courier-battery-negative-domain.
    pddl \
    courier-battery-negative-p03.pddl
; cost = 5
...
```

Battery 15, 3, −5. Fix: a precondition, same expression.

A forgotten initial value

No error: actions that read it never apply. No plan, or a strange one.

- Give a value to every fluent that is read
- Seven connections, both ways: fourteen distances

Common mistakes (2)

Facts' syntax for numbers

A value as one more argument:

```
(battery r1 100)
```

`kairos check`: *expected term, found number '100'*. An effect with =: *expected effect*. Use `(assign f e)`.

Equality on fractions

A drip adds 0.1 ml; goal `(= (dose) 0.3)`.

```
$ kairos drip-domain.pddl drip-p01.pddl
note: no plan: every state reachable from
    the initial one was searched (8 states)
     and none satisfies the goal. [
    orchestrator.why]
...
```

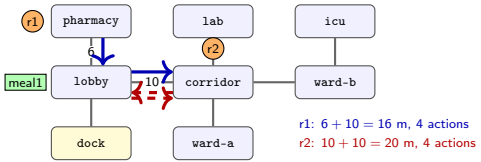
$3 \times 0.1 = 0.30000000000000004$. A range, or whole units.

1 Numeric fluents

2 Plan quality: metrics and action costs

3 Summary

When any plan is not enough




```
$ kairos courier-cost-domain.pddl
    courier-cost-p01-nometric.pddl
; cost = 4
(move r2 corridor lobby)
(pick r2 meal1 lobby)
(move r2 lobby corridor)
(drop r2 meal1 corridor)
```

- Two plans of four actions: the same plan for every model so far
- Satisficing planners stop at the first plan found
- Without a metric, cost = **plan length**
- Right only when all actions are worth the same

Action costs

```
(:requirements :strips :typing :action-costs)
(:functions
  (total-cost) - number                    ; meters so far
  (distance ?from ?to - location) - number) ; static data

(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)
               (increase (total-cost) (distance ?from ?to))))
```

```
  (= (total-cost) 0)
  (robot-at r1 pharmacy) (free r1)
  (robot-at r2 corridor) (free r2)
  (item-at meal1 lobby))
(:goal (item-at meal1 corridor))
(:metric minimize (total-cost)))
```

- Domain: a function (`total-cost`), increased by every action that costs
- The cost of `move` is read from the static `distance`
- `pick`, `drop`: no `increase`, cost zero
- Problem: starts at zero, ends with the **metric**
- The metric belongs to the problem: each problem says whether it cares

With a metric, the answer changes

```
$ kairos courier-cost-domain.pddl
    courier-cost-p01.pddl
; cost = 20
(move r2 corridor lobby)
(pick r2 meal1 lobby)
(move r2 lobby corridor)
(drop r2 meal1 corridor)
; better plan 2
; cost = 16
(move r1 pharmacy lobby)
(pick r1 meal1 lobby)
(move r1 lobby corridor)
(drop r1 meal1 corridor)
```

- The cost printed is now the value of the metric
- Kairos kept searching: `; better plan 2`
- The last plan printed is the best one

```
$ kairos validate
    courier-cost-domain.pddl
    courier-cost-p01.pddl \
    courier-cost-p01-r2.plan
plan valid, makespan 4, metric 20
```

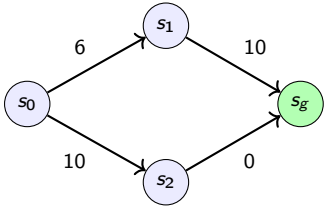
Valid: it works, not that it is good.

The requirement :action-costs

A deliberately small piece of numeric planning (PDDL 3.1, competition of 2008):

- only (total-cost) changes, a function without parameters
- only by (increase (total-cost) e), e a non-negative number or a static function of the parameters
- no precondition and no goal reads a number
- the metric is (:metric minimize (total-cost))

Kairos warns that :numeric-fluents is not declared, and plans normally.



The classical state space, finite, same states and plans: costs only weigh the transitions. A classical planner needs only a search that adds weights.

Costs from data: a kitchen with prices

```
(:types
  ingredient dish - object
  egg fat sweetener - ingredient
  omelette cake - dish)
(:action add-egg
  :parameters (?e - egg ?d - dish)
  :precondition (in-stock ?e)
  :effect (and (not (in-stock ?e)) (with-egg ?d)
               (increase (total-cost) (price ?e))))
(:action fry
  :parameters (?d - omelette)
  :precondition (and (with-egg ?d) (with-fat ?d))
  :effect (and (ready ?d) (increase (total-cost) 1)))

(:action bake
  :parameters (?d - cake)
  :precondition (and (with-egg ?d) (with-fat ?d)
                     (with-sweetener ?d))
  :effect (and (ready ?d) (increase (total-cost) 2))))
```

```
(:init
  (= (total-cost) 0)
  (in-stock farm-egg) (= (price farm-egg) 3)
  (in-stock egg1) (= (price egg1) 1)
  (in-stock egg2) (= (price egg2) 1)
  (in-stock butter1) (= (price butter1) 4)
  (in-stock oil1) (= (price oil1) 2)
  (in-stock oil2) (= (price oil2) 2)
  (in-stock honey1) (= (price honey1) 5)
  (in-stock sugar1) (= (price sugar1) 1))
(:goal (and (ready om1) (ready cake1)))
(:metric minimize (total-cost)))
```

- A cost from data, `(price ?e)`, or a constant, the gas
- Types do the work of recipes: any `fat` will do

The cheapest meal

```
$ kairos kitchen-domain.pddl
    kitchen-p01.pddl
; cost = 18
(add-egg farm-egg cake1)
(add-egg egg1 om1)
(add-fat butter1 cake1)
(add-fat oil1 om1)
(add-sweetener honey1 cake1)
(bake cake1)
(fry om1)
```

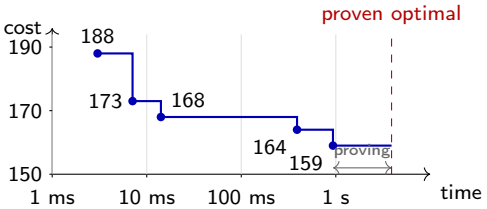
Seven actions each: length cannot tell them apart.

```
; better plan 2
; cost = 10
(add-fat oil1 cake1)
(add-fat oil2 om1)
(add-egg egg1 cake1)
(add-egg egg2 om1)
(add-sweetener sugar1 cake1)
(bake cake1)
(fry om1)
```

Zero-cost actions: free for the search. In the world?

The planner at work: anytime search

- **Optimal**: rules out every cheaper plan before answering
- **Anytime**: a first plan fast, then strictly better ones
- Four deliveries, two robots (p02)




```
$ kairos courier-cost-domain.pddl courier-cost-p02.pddl -v 2>&1 \
    >/dev/null | grep -E "plan [0-9]|^kairos: solved"
[   0.003s] [search] plan 0 with 29 actions, cost 188
...
[   0.944s] [search] plan 4 with 28 actions, cost 159
kairos: solved (proven optimal), cost 159
```

What proven optimal means

```
(move r1 dock lobby)
(move r1 lobby pharmacy)
(pick r1 meds1 pharmacy)
...
(move r1 corridor ward-b)
(move r1 ward-b icu)
(drop r1 meds2 icu)
```

The best plan, 159 m (`courier-cost-p02.plan`).

- No plan of 158 meters exists **in the model, for the metric as written**
- Every action is of `r1`; `r2` never leaves the ICU
- Right for meters: each meter of `r2` adds to the sum
- To finish early the model needs time (Part IV); to spread the wear, another measure
- An optimal plan answers the question that was written

Good enough

```
$ kairos courier-cost-domain.pddl courier-cost-p02.pddl \
    --no-anytime -v 2>&1 | grep -E "cost =|^kairos: solved"
; cost = 188
kairos: solved (no improvement), cost 188
```

Above, the first plan only. `--target-cost C` stops at cost `C` or less (14 ms):

```
$ kairos courier-cost-domain.pddl courier-cost-p02.pddl \
    --target-cost 170 -v 2>&1 | grep -E "cost =|^kairos: solved"
; cost = 188
; cost = 173
; cost = 168
kairos: solved (target reached), cost 168
```

A budget, and what the status words mean

`--timeout S`: a budget in seconds; with half a second:

```
$ kairos courier-cost-domain.pddl courier-cost-p02.pddl \
    --timeout 0.5 -v 2>&1 | grep -E "cost =|^kairos: solved"
; cost = 188
...
kairos: solved (timeout), cost 164
```

- Always `solved`; only `proven optimal` is a guarantee
- **Optimal** planners promise minimal cost; **satisficing** ones, a plan

Status	Meaning
`proven optimal`	nothing better exists
`target reached`	good enough by `C`
`timeout`	best found in time
`no improvement`	first plan only

Metrics over any fluent

```
(:functions
  (distance ?from ?to - location)  ; meters, static
  (consumption ?r - robot)         ; energy per meter, static
  (meters)                         ; covered by all the robots
  (energy-used))                   ; used by all the robots

(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to))
  :effect (and
    (not (robot-at ?r ?from)) (robot-at ?r ?to)
    (increase (meters) (distance ?from ?to))
    (increase (energy-used)
              (* (distance ?from ?to) (consumption ?r)))))
```

```
(:metric minimize (meters)))
(:metric minimize (energy-used)))
```

- The metric can be any expression over numeric fluents
- Each measure in its own **counter**; each problem chooses
- Breaks `:action-costs` twice: two fluents change, one by a product
- No precondition reads them
- `r1` light (1) at the dock; `r2` heavy (3) in ward B

Meters or energy

```
$ kairos courier-quality-domain.pddl
    courier-quality-meters.pddl
; cost = 68
...
(drop r1 meds1 ward-a)
```

- Meters: each robot takes the delivery near it (30 m, 38 m)
- Energy: 128, 92, 88: longer plans, cheaper

```
$ kairos courier-quality-domain.pddl courier-quality-energy.pddl \
    -v 2>&1 >/dev/null | grep -E "plan [0-9]|^kairos: solved"
[   0.002s] [search] plan 0 with 13 actions, cost 128
...
kairos: solved (proven optimal), cost 88
```

The optimizer finds the loopholes

```
(move r1 lobby corridor)
(drop r1 meds1 corridor)
(move r1 corridor ward-b)
...
(drop r1 sample1 lab)
(move r1 lab corridor)
(pick r1 meds1 corridor)
(move r1 corridor ward-a)
(drop r1 meds1 ward-a)
```

- We expected 92; the planner saved 4 units
- It leaves the medicines on the corridor floor and enters ward A only once
- Valid in the model. May medicines wait unattended? A question for the model card
- If not: a precondition, or the constraints of chapter 15

Read the best plan with suspicion

An optimizing planner searches the corners of a model and finds what it forgot to forbid.

Maximizing: (:metric maximize (battery r1))

```
$ kairos courier-battery-domain.pddl courier-battery-max.pddl
; cost = 62
...
note: the search minimizes the number of actions; the metric of the problem is
     reported for each plan but does not guide the search yet [orchestrator.
    metric]
```

```
$ kairos validate courier-battery-domain.pddl \
    courier-battery-max.pddl courier-battery-max.
    plan
plan valid, makespan 11, metric 100
```

Advice

Use **sums of things that only accumulate**.

Several wishes in one number

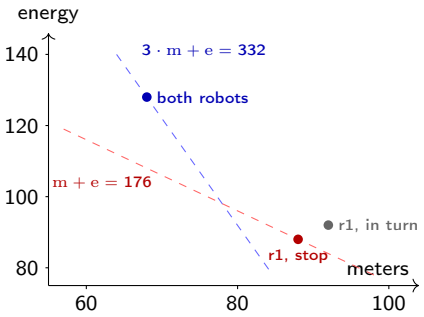
Weighted sum: a meter weighs three units of energy (`mix1`):

```
(:metric minimize (+ (* 3 (meters)) (energy-used)))
```

Best: both robots, $3 \times 68 + 128 = 332$. With `(+ (meters) (energy-used))`:

```
$ kairos courier-quality-domain.pddl
      courier-quality-mix2.pddl \
    | grep cost
; cost = 196
; cost = 176
```

`r1` with its stop: $88 + 88 = 176$ wins.

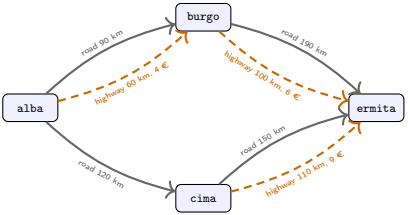


The ratio of the weights chooses the point. With money as unit, weights are prices.

A road trip: tolls and fuel

```
(:action drive-highway
  :parameters (?c - car ?from ?to - town)
  :precondition (and (car-at ?c ?from) (highway ?from ?to))
  :effect (and
    (not (car-at ?c ?from)) (car-at ?c ?to)
    (increase (tolls-paid) (toll ?from ?to))
    (increase (fuel-used)
              (/ (highway-km ?from ?to) (km-per-liter ?c))))))
```

```
; money: the tolls plus the fuel at 2 euros per liter
(:metric minimize (+ (tolls-paid) (* 2 (fuel-used))))
```




```
$ kairos road-trip-domain.pddl
    road-trip-p01.pddl
; cost = 27
(drive-road car1 alba cima)
(drive-road car1 cima ermita)
; better plan 2
; cost = 25
(drive-road car1 alba burgo)
(drive-highway car1 burgo
    ermita)
```

Tunnel: 9 € of fuel saved for 6 of toll. A van (p02): both highways.

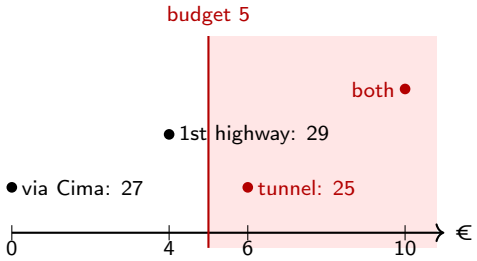
A limit or a metric

A **hard limit** splits plans; a metric orders them.

```
:precondition (and
  (car-at ?c ?from) (highway ?from ?to)
  (<= (+ (tolls-paid) (toll ?from ?to)) (toll-budget)))
```

```
$ kairos road-trip-budget-domain.
    pddl road-trip-budget-p01.pddl
; cost = 27
(drive-road car1 alba cima)
(drive-road car1 cima ermita)
```

Budget 5 €: the plan of 25 needs a toll of 6. A limit only takes plans away.



- Both highways: invalid, `10 <= 5 is false`
- Failure: a **limit**. Only worse: a **metric**

The MODEL method: what makes a plan better

Step M: two questions about quality

What makes one plan better than another? Name the measure and its unit: meters, euros, energy, handlings. If "nothing", say so: no metric.
Is it one number? A common unit makes the weights prices; otherwise somebody decides how much of one is worth a unit of another. Write the weights with their reason. And for each wish: a preference (metric) or a condition of success (limit)?

- Card: *Success* gains "the fewer meters the better"; *Assumptions*: picking is free, a robot that stands still costs nothing
- Step O: each measure a counter, dynamic, from zero, read by no precondition unless a limit
- Step D: a cost column, zeros justified. Step E: two plans of known cost; a forgotten cost

Alternatives: three ways to say "better"

	Plan length	Action costs	Fluent metric
Fidelity	Right only if all actions are worth the same. Here: 13 actions, 68 m, 128 of energy	One sum of fixed costs per action. Here: 68 m, optimal	Any expression: energy per robot, weighted sums. Here: 88 of energy, or the mixes
Size	92 ground actions, no fluent	92, one fluent no condition reads	92, two counter fluents
Guidance	Classical search	Classical space with weights; best supported	Good while counters grow by fixed amounts
Clarity	Nothing to write, nothing said	One line per action, one in the problem	Counters to maintain; the metric documents the weights
Portability	Every planner	`:action-costs`: classical planners too	`:numeric-fluents`: numeric planners only

Verdict. Nothing when nothing matters. Action costs when quality is a sum of fixed costs of actions: most of the time. Fluent metrics for state-dependent costs, measures weighed per problem, or a counter that is also a limit.

Alternatives: a limit or a metric

	Metric only	Limit only	Limit and metric
Fidelity	No plan lost; may spend more than the driver has. Best: 25 €, toll 6	Never over the budget; any plan within it, good or bad, maybe none	Never over, and the cheapest within. Best: 27 €
Size	Counters only	One numeric precondition per limited action	Both
Guidance	Costs guide the search	Dead ends: a toll paid early can block the only way later	Both effects
Clarity	The trade-off is in the weights	The rule where it applies, in the action	Two places to read
Portability	Fluent metric	Numeric preconditions	Both

Verdict. Not rivals: the limit for what must not happen, the metric for what should be small. When in doubt, start with the metric alone and read the best plan; if it does something unacceptable, a limit was missing.

Common mistakes (1)

An action that forgot its cost

`add-fat` without `increase`: free. Best meal 6, with butter.

```
$ kairos kitchen-free-domain.pddl
    kitchen-free-p01.pddl
...
; cost = 6
...
(add-fat butter1 cake1)
```

Fix: a cost for every action, zeros written.

The metric in the wrong file

`(:metric ...)` in the domain: a syntax error.

```
$ kairos check
    kitchen-wrong-domain.pddl
kitchen-wrong-domain.pddl:22:4:
    error: unknown domain section
    :metric [parse.unknown_section
    ]
invalid
```

The reverse: no error.

Common mistakes (2)

Negative costs and endless improvement

A refund, or `maximize` over something that grows for ever. Kairos stops optimizing and says so (sugar at -1: first plan, 18). A loop with negative total cost: no best plan exists. Fix: non-negative costs; bound what is maximized.

The initial value of the cost

Without `(= (total-cost) 0)` the fluent is undefined. Kairos is lenient; a planner that follows the rule finds no costly action applicable. Initialize every counter.

Trusting the first plan, or the wrong word

"The model sends the far robot", when a later plan does not: in the round, 188 first, 159 best. Read the last plan and the status line with `-v`. The anytime output saved to one file is not a valid plan: use `--output` and validate the last file.

1 Numeric fluents

2 Plan quality: metrics and action costs

3 Summary

What to take away

Numeric fluents

- Functions put numbers in the state; expressions read the state before the action
- Static functions are folded into ground actions; dynamic ones are state
- Undefined is not zero: silent missing actions
- Unbounded numbers: no proof of "no plan"
- Step O: quantity? threshold? bounded?

Plan quality

- No metric: cost is plan length
- Action costs: portable, classical state space
- Fluent metrics: counters and weighted sums
- Anytime: read the last plan and the status
- Step M: what makes a plan better? one number? limit or metric?

Exercises to try

Chapter 8

- 8.3.1: predict the plan of p02 with 50 units
- 8.3.2: a reserve of 10 units
- 8.5.1: split the wine with jugs of 8, 5 and 3
- 8.6.1: delete the weight of the meal
- 8.7.2: the endless truck with 1, 2, 4 seconds
- 8.8.3: the watering robot, steps M, O, D, E

Chapter 9

- 9.2.1: a nurse for every pick and drop
- 9.4.2: make an optimal plan use both robots
- 9.5.1: energy plus 5 per handling
- 9.6.2: the fuel price at which both highways pay
- 9.7.1: the fuel of the trip as a hard limit
- 9.8.2: the drone, battery and risk