Appendix

Modelling Examples in PDDL and UP

This appendix presents detailed modelling examples comparing the PDDL encoding with the new Unified Planning (UP) syntax using the proposed extensions. We include representative fragments for the selected domains to illustrate the advantages in terms of conciseness, clarity, and structure. These examples complement the conceptual explanations in the main paper and demonstrate how the presented features simplify repetitive encodings.

We focus on two running examples throughout this appendix: n-Puzzle, a classic sliding tile puzzle where numbered tiles must be moved into a goal configuration on a grid; and Plotting, a puzzle video game in which the goal is to eliminate blocks from a grid until only a fixed number remain.

We illustrate each extension with code fragments from both the UP models using our extensions and the handcrafted PDDL encodings.

ArrayType

We begin with the modelling of the 8-Puzzle to illustrate the use of `ArrayType`. Fig 1 shows the initial and goal states for a specific instance of the puzzle on a 3×3 grid.

Without arrays, one must explicitly define all adjacency relationships (`inc`, `dec`) between individual cells, resulting in a long, repetitive, and error-prone model. While this may still be manageable in small instances like the one shown, maintaining and initialising these predicates becomes increasingly tedious as the grid size increases.

Listing 1 shows part of the PDDL domain and Listing 2 illustrates the corresponding problem file.

```
(:predicates
    (tile ?x)
    (position ?x)
    (at ?t ?x ?y)
    (blank ?x ?y)
    (inc ?p ?pp)
    (dec ?p ?pp))
```

Listing 1. 8-Puzzle PDDL domain fragment

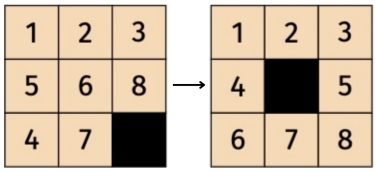


Fig. 1. Initial and goal states of an 8-Puzzle instance.

In contrast, arrays make these relationships implicit. The grid can be naturally represented as a two-dimensional fluent, as shown in Listing 3, allowing both the initial and goal states to be expressed as simple 2D arrays. The default initial value of the fluent must match the type of the innermost elements in the array structure. Initial values can also be specified row-by-row or element-by-element.

```
(:objects
    t1 t2 t3 t4 t5 t6 t7 t8
    p1 p2 p3)
(:init
    (tile t1) (tile t2) (tile t3) (tile t4)
    (tile t5) (tile t6) (tile t7) (tile t8)
    (position p1) (position p2) (position p3)
    (inc p1 p2) (inc p2 p3)
    (dec p3 p2) (dec p2 p1)
    (at t1 p1 p1) (at t2 p1 p2) (at t3 p1 p3)
    (at t5 p2 p1) (at t6 p2 p2) (at t8 p2 p3)
    (at t4 p3 p1) (at t7 p3 p2) (blank p3 p3))
(:goal
    (and
        (at t1 p1 p1) (at t2 p1 p2) (at t3 p1 p3)
        (at t4 p2 p1) (blank p2 p2) (at t5 p2 p3)
        (at t6 p3 p1) (at t7 p3 p2) (at t8 p3 p3)))
```

Listing 2. 8-Puzzle PDDL problem fragment

```
puzzle = Problem("puzzle-problem")
# Declaring types, objects and fluents
Tile = UserType("Tile")
t0 = Object("t0", Tile)
t1 = Object("t1", Tile)
t2 = Object("t2", Tile)
t3 = Object("t3", Tile)
t4 = Object("t4", Tile)
t5 = Object("t5", Tile)
t6 = Object("t6", Tile)
t7 = Object("t7", Tile)
t8 = Object("t8", Tile)
at = Fluent("at", ArrayType(3, ArrayType(3, Tile)))
# Adding fluent with default value
puzzle.add_fluent(at, default_initial_value=t0)
# Specifying the initial and goal state
puzzle.set_initial_value(at,
    [t1,t2,t3], [t5,t6,t8], [t4,t7,t0]])
puzzle.add_goal(at,
    [[t1,t2,t3], [t4,t0,t5], [t6,t7,t8]])
```

Listing 3. 8-Puzzle UP representation using arrays.

Undefinedness: To showcase the use of undefined positions, we consider a variation of the classical 8-Puzzle where the top-right tile is replaced by a wall. This results in a 3×3 board with only seven tiles and one immovable position at `(0,2)`. To model this, we define the `at` fluent using the `undefined_positions` parameter to mark that cell as inaccessible, as shown in Listing 4.

```
at = Fluent("at", ArrayType(3, ArrayType(3, Tile)),
            undefined_positions=[(0,2)])
```

Listing 4. 8-Puzzle UP representation using arrays with an undefined position.

Integer-Type Parameters in Actions

Continuing with the 8-Puzzle problem, four actions are defined for each movement direction. A PDDL version of the `move_right` action can be seen in Listing 5. In this encoding, the indices of the array, the value stored at that position, and an additional parameter to denote the next column (the position of the adjacent empty cell) must be explicitly included as parameters. This is required because the precondition needs to pin down the correct tile value at the specific position so that it can be used in the effects.

```
(:action move-right
  :parameters (?r ?c ?nc - idx ?t - tile)
  :precondition (and
    (grid ?t ?r ?c)
    (next ?nc ?c)
    (empty ?r ?nc))
  :effect (and
    (not(empty ?r ?nc))
    (not(grid ?t ?r ?c))
    (empty ?r ?c)
    (grid ?t ?r ?nc)))
```

Listing 5. 8-Puzzle PDDL `move-right` action

To overcome this limitation, we introduce support for bounded integer parameters in actions, allowing users to treat integers as such rather than encoding them as objects. This is remarkably useful when working with arrays, as it allows for direct indexing of elements. Moreover, since arithmetic operations are also supported, one can naturally refer to, for instance, adjacent cells. As shown in Listing 6, the `move-right` action can now be expressed more naturally, with the row and column treated as bounded integers. Since moving right is only applicable in one of the first two columns, we can declare the column parameter `c` as the bounded integer `IntType(0,1)`, preventing movement beyond the rightmost column. In any case, if we had used `IntType(0,2)` instead, our permissive array treatment mode of undefinedness would deal with the out-of-bound accesses, see Section **??**.

```
mr = InstantaneousAction("move-right",
                         r=IntType(0,2), c=IntType(0,1))
c = mr.parameter("c")
r = mr.parameter("r")
mr.add_precondition(Equals(grid[r][c+1], t0))
mr.add_effect(grid[r][c+1], grid[r][c])
mr.add_effect(grid[r][c], t0)
```

Listing 6. 8-Puzzle `move-right` action using Integer Parameters

This leads to simpler and more intuitive action definitions.

Count Expression

We illustrate the use of the `Count` expression with the Plotting domain (Fig. 2).

In this model, the fluent `blocks` is a 2D array of `Colour` values indexed by integers representing `rows` and `columns`, as shown in Listing 7.

The goal condition, shown in Listing **??**, ensures that the number of non-empty cells (i.e., those whose `Colour` is not

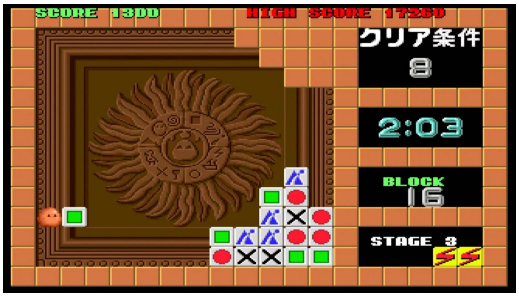


Fig. 2. Plotting (Taito, 1989). The goal is to reduce the number of blocks in the grid to a target number (8 here) or fewer. The avatar (left) shoots blocks into the grid. If the shot block hits one of the same pattern, that block is removed. State changes are complex because multiple blocks may be removed in a single shot and gravity affects the blocks in the grid.

`N`) does not exceed a given threshold (`2` in this case). Python list comprehensions provide a convenient way to generate the set of Boolean conditions used in the count expression

```
plotting = Problem
Colour = UserType("Colour")
R = Object("R", Colour)
N = Object("N", Colour) % None
blocks = Fluent("blocks", ArrayType(5,ArrayType(5,Colour)))
...
plotting.add_goal(
  LE(
    Count(
      [Not(Equals(blocks[i][j], N))
        for i in range(5)
        for j in range(5)]),
    2))
```

Listing 7. Plotting UP goal condition using Count.

This condition would be particularly cumbersome to express in PDDL, where grid positions are also simulated using objects of a user-defined type (`number`), as shown in Listing 8. This requires explicitly declaring all possible positions and defining their relationships (e.g., `succ`, `pred`, `lt`, `gt`) as fluents.

Listing 9 shows the goal condition for this same instance, which ensures that all possible combinations of positions that could contain a block are distinct and that no other cell is coloured. This approach becomes increasingly complex with larger grids or thresholds.

```
(:objects
  n1 n2 n3 n4 n5 - number
  c1 c2 c3 c4 - colour)
(:init
  (coloured n1 n1 c1) (coloured n1 n2 c1) ...
  (succ n1 n2) (succ n2 n3) ...
  (lt n1 n2) (lt n1 n3) ...
  (distance n1 n2 n1) (distance n1 n3 n2) ...
  (istoprow n1) (isbottomrow n5) ...
)
```

Listing 8. Fragment of PDDL initial state for Plotting.

```
(:goal
  (exists (?x1 ?y1 ?x2 ?y2 ?x3 ?y3 ?x4 ?y4 - number)
    (and
      (or (not (= ?x1 ?x2)) (not (= ?y1 ?y2)))
      (or (not (= ?x1 ?x3)) (not (= ?y1 ?y3)))
      (or (not (= ?x1 ?x4)) (not (= ?y1 ?y4)))
      (or (not (= ?x2 ?x3)) (not (= ?y2 ?y3)))
      (or (not (= ?x2 ?x4)) (not (= ?y2 ?y4)))
      (or (not (= ?x3 ?x4)) (not (= ?y3 ?y4)))
      (forall (?x5 ?y5 - number) (or
        (and (= ?x1 ?x5) (= ?y1 ?y5))
        (and (= ?x2 ?x5) (= ?y2 ?y5))
        (and (= ?x3 ?x5) (= ?y3 ?y5))
        (and (= ?x4 ?x5) (= ?y4 ?y5))
        (coloured ?x5 ?y5 N)))))
)
```

Listing 9. Equivalent goal condition in PDDL.

Integer Range Variables

Continuing with the Plotting domain, player actions to remove blocks involve shooting the block held by the avatar toward a selected row or column. All identical blocks that the shot collides are removed until a distinct block or a wall is reached. Complex effects may occur due to gravity applied to the remaining blocks.

The domain has several shooting actions depending on the conditions of the shot. The `shoot-partial-row` action, for instance, requires that all blocks in row `r` up to column `t` are either of a specific colour `c` or empty, and that at least one of them has colour `c`.

These preconditions can be compactly expressed using quantifiers over a `RangeVariable`, as illustrated in Listing 10, where the quantified variable (`b`) ranges from `0` to `t`.

Listing 10. Plotting `shoot-partial-row` action using Range Variables.

```
spr = InstantaneousAction("shoot-partial-row", c=Colour,
                          r=IntType(0, rows-1),
                          t=IntType(0, columns-2))
c = spr.parameter("c")
r = spr.parameter("r")
t = spr.parameter("t")
b = RangeVariable("b", 0, t)
# condition 1
spr.add_precondition(Forall(Or(Equals(blocks[r][b], c),
     Equals(blocks[r][b], N)), b))
# condition 2
spr.add_precondition(Exists(Equals(blocks[r][b], c), b))
```

Compilation Times

The compilation time added by our extensions is negligible in most cases. The only exception is when using the `Count` expression in hard instances, where conditions become combinatorially large.

A summary of the observed compilation times:

- **Pancake:** Compilation remains under 4 seconds.
- **Plotting:** Around 10% of instances take more than 5 seconds, mainly due to combinatorial conditions involving `Count`.
- **15-Puzzle:** All instances compile in under 2 seconds.
- **Sokoban:** Compilation takes under 5 seconds.
- **Puzznic:** Compilation remains below 7 seconds.
- **Rush Hour:** Compilation never exceeds 2 seconds.

These compilation times are small compared to the overall solving time, and we believe the added expressiveness clearly justifies the compilation effort. Even in the most complex domains, the overhead remains acceptable.

Plots

This section presents additional plots illustrating the performance of different compilation strategies across the problems introduced in the main paper. Colours indicate different compilation versions, while markers distinguish solvers: Fast Downward (F), Symk (S) and ENHSP (E).

Table I defines the distinct compilation strategies used, i.e. the compiler sequences. The compilers' acronyms used are the following: IntParameterActionsRemover (IPAR), ArraysRemover (AR), ArraysLogarithmicRemover (ALR), CountRemover (CR), CountIntRemover (CIR), IntegersRemover (IR), and UserTypesFluentRemover (UFR).

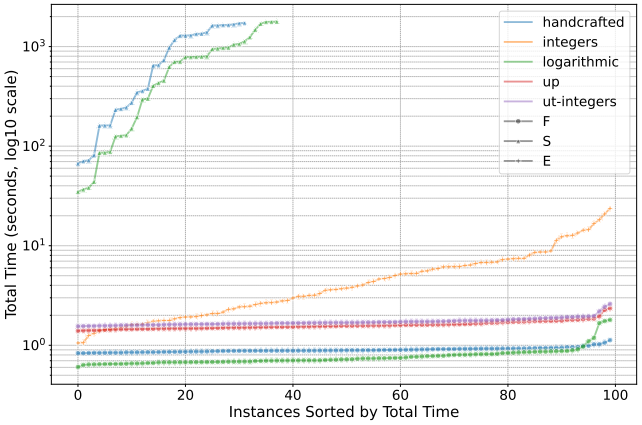
Compilation Strategies	**Compilers Sequence**
up	*IPAR*, *AR*, *UFR*
logarithmic	*IPAR*, *ALR*
ut-integers	*IPAR*, *AR*, *IR*, *UFR*
count	*IPAR*, *AR*, *CR*, *UFR*
count-int	*IPAR*, *AR*, *CIR*, *IR*, *UFR*
count-int-num	*IPAR*, *AR*, *CIR*, *UFR*
integers	*IPAR*, *AR*

TABLE I
COMPILATION STRATEGIES WITH THEIR COMPILERS APPLICATION SEQUENCE. ALL STRATEGIES ARE COMPATIBLE WITH CLASSICAL PLANNERS, EXCEPT FOR *count-int-num* AND *integers*, WHICH PRESERVE NUMERIC FLUENTS AND THEREFORE REQUIRE A NUMERIC PLANNER.

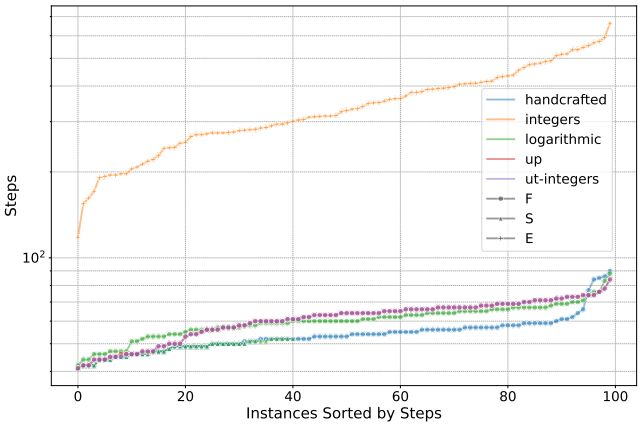
Each following section corresponds to a specific domain, with plots grouped by solving mode: OneShot, Anytime, and Optimal.

15-PUZZLE

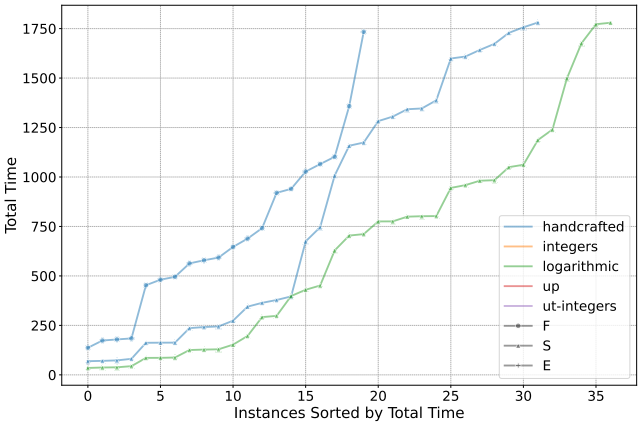
OneShot: 15-Puzzle — Total Time by Compilation and Solving.



Anytime: 15-Puzzle — Action Steps by Compilation and Solving.

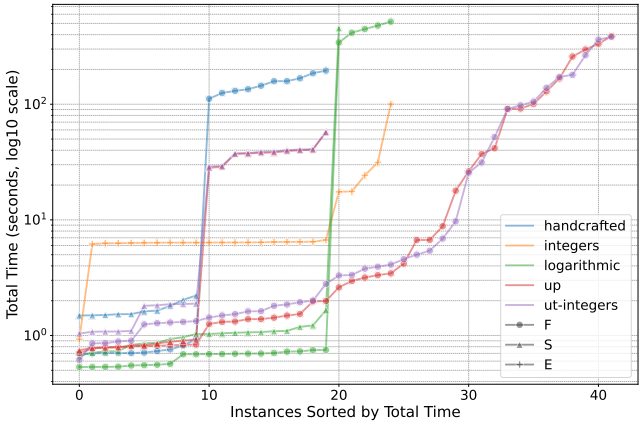


Optimal 15-Puzzle — Total Time by Compilation and Solving.

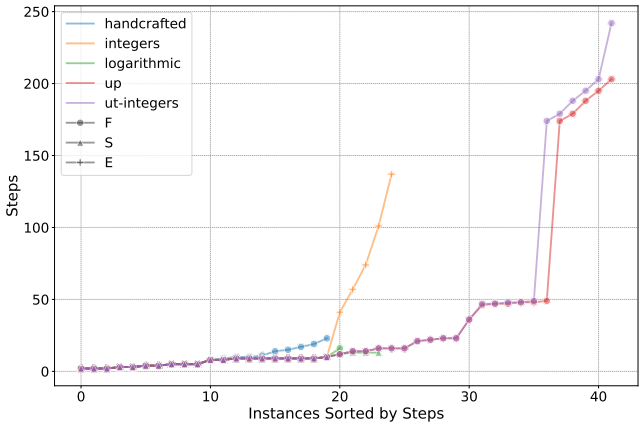


PANCAKE

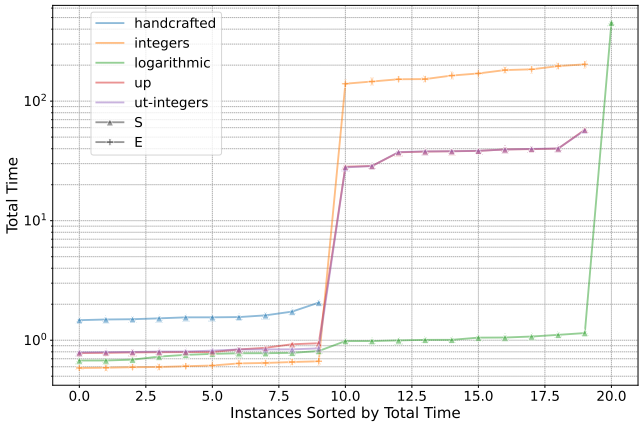
OneShot: Pancake — Total Time by Compilation and Solving.



Anytime: Pancake — Action Steps by Compilation and Solving.

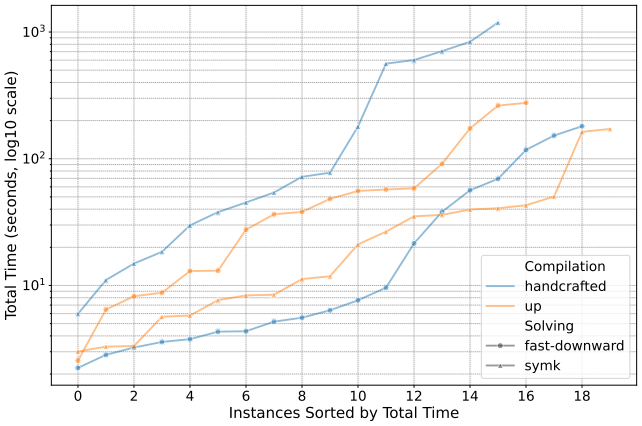


Optimal Pancake — Total Time by Compilation and Solving.

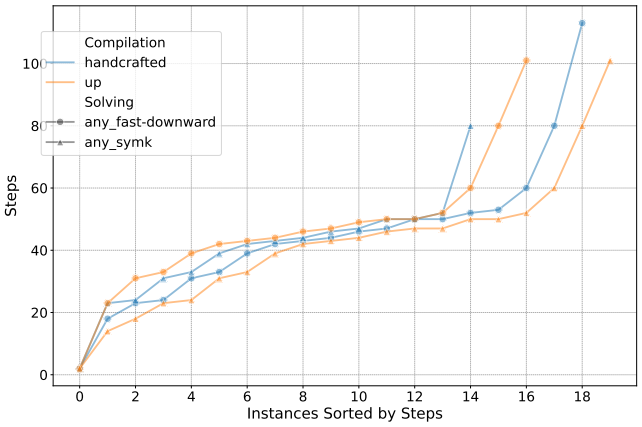


Sokoban

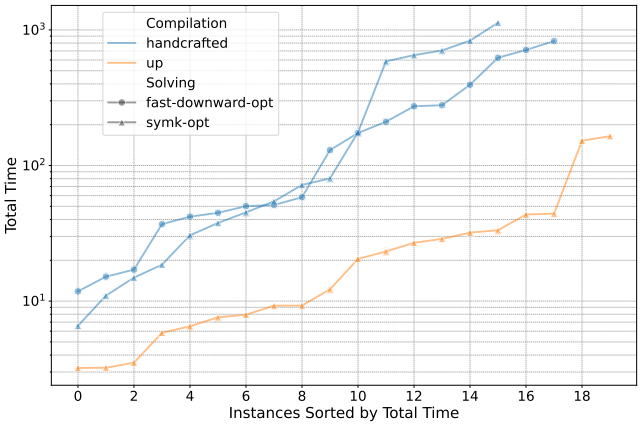
OneShot: Sokoban — Total Time by Compilation and Solving.



Anytime: Sokoban — Action Steps by Compilation and Solving.

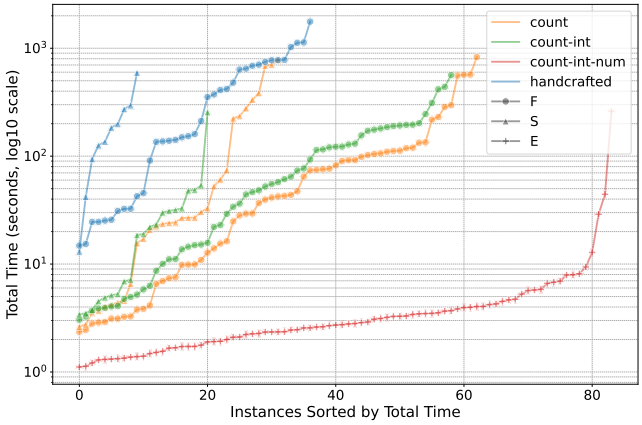


Optimal Sokoban — Total Time by Compilation and Solving.

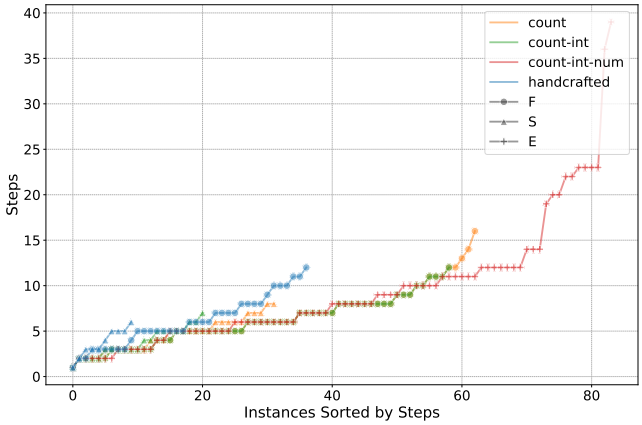


Plotting

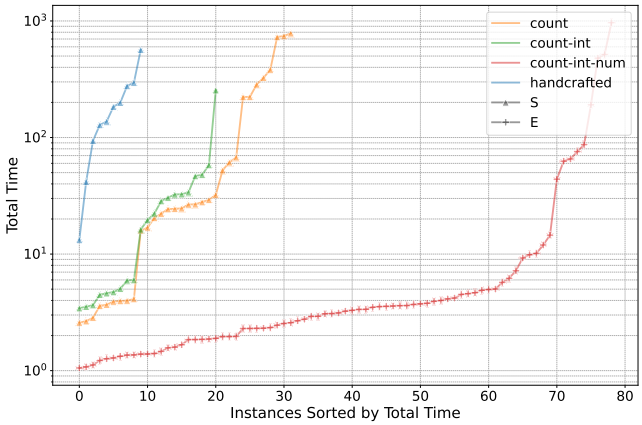
OneShot: Plotting — Total Time by Compilation and Solving.



Anytime: Plotting — Action Steps by Compilation and Solving.

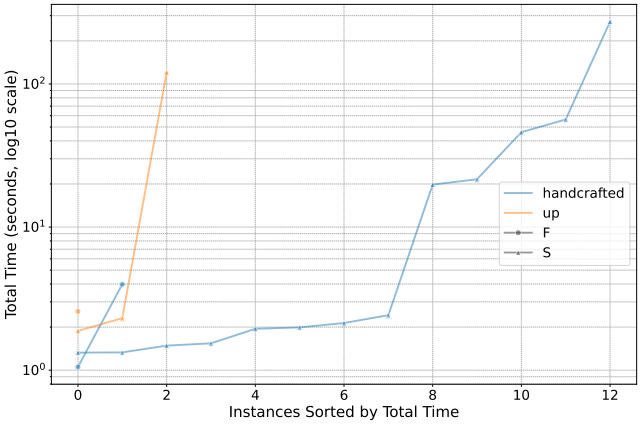


Optimal Plotting — Total Time by Compilation and Solving.

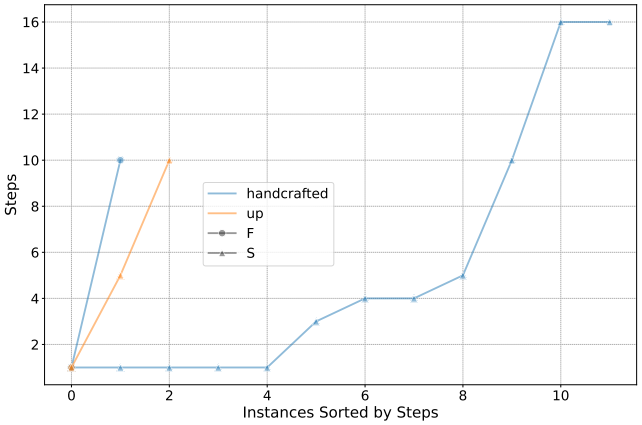


PUZZNIC

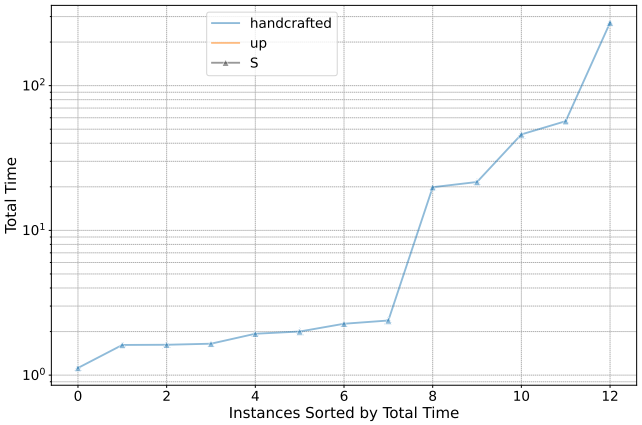
OneShot: Puzznic — Total Time by Compilation and Solving.



Anytime: Puzznic — Action Steps by Compilation and Solving.

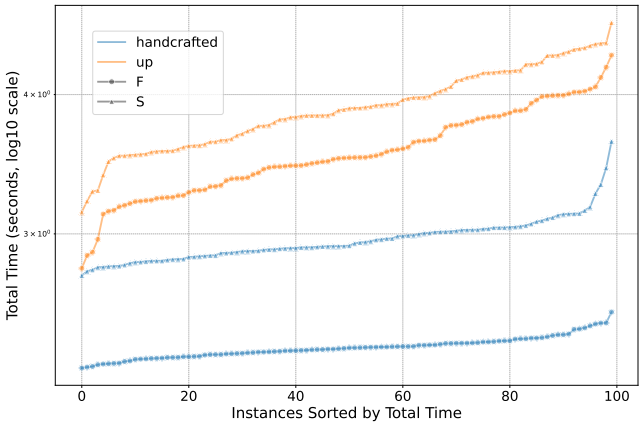


Optimal Puzznic — Total Time by Compilation and Solving.

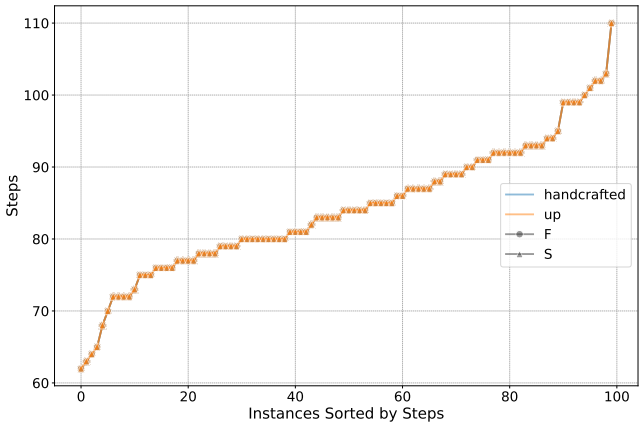


RUSH HOUR

OneShot: Rush Hour — Total Time by Compilation and Solving.



Anytime: Rush Hour — Action Steps by Compilation and Solving.



Optimal Rush Hour — Total Time by Compilation and Solving.