The Jumpbot Domain for Numeric Planning

Johannes Aldinger
University of Freiburg, Germany
aldinger@informatik.uni-freiburg.de

Johannes Löhr
Airbus Defence and Space, Immenstaad, Germany
johannes.loehr@airbus.com

Abstract

The `Jumpbot` domain for numeric planning with instantaneous actions models a walking robot that has to reach a target region by jumping over water ditches. The kinematic of the robot is modeled by its current position and velocity vector. The planner has to reason about the correct accelerations, rotations, velocities, jump positions and space for the deceleration. This domain description provides a set of benchmark instances for numeric planning in an area which is underrepresented by prevailing benchmarks: the use of numeric variables to model physical properties as opposed to their use for modeling resources.

Introduction

The `Jumpbot` domain offers a set of 20 benchmark instances for numeric planning with discrete actions modeled in `PDDL2.1, layer 2` (Fox and Long 2003). The numeric quantities are used to model physical quantities for a simple walking robot scenario. The states are differentially coupled "footprints" of the robot that follow the robot's kinematics. The planning problem consists of four numeric variables: x, y, v_x, v_y which are used to describe position and velocity of the robot. The task is to plan the step trace from an initial position to a target region within a world as shown in Figure 1: We depict the initial state by a red cross, together with a velocity vector symbolizing the current orientation and speed of the robot. Water is depicted by blue waves and the solid ground by white surface. The robot is only allowed to step on solid ground, but it may step or jump over water ditches. The domain contains information about the robot's mass, velocity, momentum and acceleration capacities. The planner can use this information to generate solutions to reach the goal by accelerating and jumping over the water areas. This requires reasoning about the correct accelerations and velocities, a correct jump position, and enough space for the deceleration.

Numbers are mainly either used to express the available quantity of a certain resource, or to describe physical quantities such as position, speed or voltage. As the majority of current benchmark problems focus on the use of numeric variables to model resources, the `Jumpbot` domain fills a gap by modeling a coupled system of linear differential equations which arise in many real world problems. In the next section we cover the physical background of the modeled problem. Afterwards, we illustrate the 20 benchmark instances of the `Jumpbot` domain. Finally, we give advice on how to create new `Jumpbot` instances.

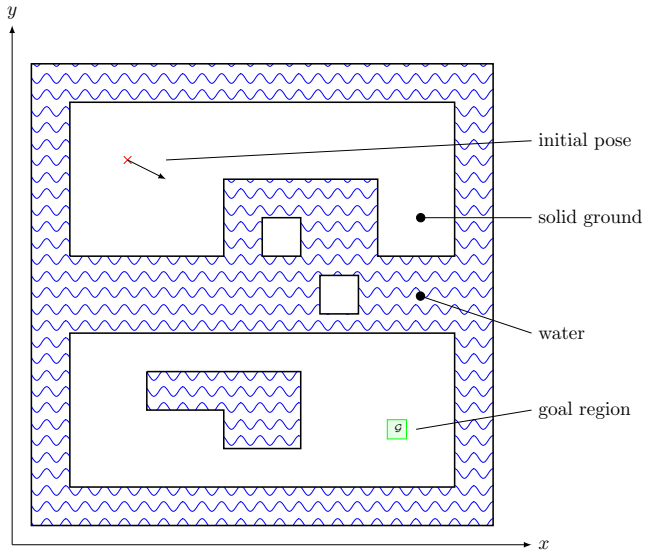

Figure 1: Scenario description of `Jumpbot` instances

Physical Foundation of the `Jumpbot` Domain

The numeric variables in the `Jumpbot` domain model linear differential equations. A switched hybrid system captures the evolution of a numeric state $\mathbf{s}(t)$ over time. The system, in our scenario the robot, can switch between *modes* where different physical forces take effect. In the case of planning, applying an action implies that the system is switched to the corresponding mode. The evolution of a numeric state under a given mode can be expressed by

$$\mathbf{s}(t+\delta) = \Phi_m(\delta)\mathbf{s}(t) + \Psi_m(\delta), \qquad (1)$$

where $\mathbf{s}(t)$ is the vector of numeric state variables, m is a mode, δ the duration for which the mode is applied, $\Phi_m(\delta)$ a *state transition matrix* and Ψ_m a vector of state-independent external influences. For numeric planning with instantaneous actions, the duration of an action is determined in advance, and we are only interested in the states in between

mode switches (in between actions). The predetermined duration of the actions in the `Jumpbot` domain is still represented as action cost. The evolution of a state at time step k with fixed actions is then

$$\mathbf{s}_{k+1} = \Phi_a \mathbf{s}_k + \Psi_a. \tag{2}$$

The matrix Φ_a and vector Ψ_a are the homogeneous and inhomogeneous solutions of the underlying continuous dynamics, and can be computed from time invariant dynamic matrices A_m. While the required operations are expensive they can be precomputed once when creating the planning domain (cf. Löhr (2014) for more details). During planning, only matrix sum and product have to be computed.

The `Jumpbot` domain contains six actions: `step`, `jump`, `accelerate`, `decelerate`, `steer left` and `steer right`. All actions (except `jump`) have a duration δ of 0.5 seconds. The `jump` action is modeled to get over obstacles in the current direction and has a step length of $\delta = 1$ second. Logical dependencies of the action sequences are also involved. The `steer right` action cannot be followed by a `steer left` action and vice versa. There has to be at least one step into the current direction before turning to the opposite direction is allowed. Moreover, the `jump` action can only be executed if the velocity in x or y direction is larger than 2. The dynamics of the robot are captured by a switched hybrid system. For the `step` and the `jump` action a simple double integrator model is used. The velocities are the time derivatives of the position in x and y direction are given by

$$\dot{x} = v_x \qquad \text{and} \qquad \dot{y} = v_y. \tag{3}$$

The robot can accelerate in the current direction by activation of both state feedback controllers

$$\dot{v}_x = k_{\text{acc}} v_x \qquad \text{and} \qquad \dot{v}_y = k_{\text{acc}} v_y \tag{4}$$

and decelerate by the converse activation of both state feedback controllers

$$\dot{v}_x = -k_{\text{dec}} v_x \qquad \text{and} \qquad \dot{v}_y = -k_{\text{dec}} v_y. \tag{5}$$

The robot can change the velocity vector towards the right hand side by the application of the steering control laws

$$\dot{v}_x = k_{\text{steer}} v_y \qquad \text{and} \qquad \dot{v}_y = -k_{\text{steer}} v_x \tag{6}$$

and to the left hand side by the application of the steering control laws

$$\dot{v}_x = -k_{\text{steer}} v_y \qquad \text{and} \qquad \dot{v}_y = k_{\text{steer}} v_x. \tag{7}$$

Equation 3 to 7 yield the state space differential system $\dot{\mathbf{s}} = A_m \mathbf{s}$, where $\mathbf{s} = (x, y, v_x, v_y)^\intercal$ and $A_m \in \{A_{\text{step}}, A_{\text{jump}}, A_{\text{acc}}, A_{\text{dec}}, A_{\text{right}}, A_{\text{left}}\}$. The dynamic matrices are given by

$$A_{\text{step}} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad A_{\text{jump}} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A_{\text{acc}} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & k_{\text{acc}} & 0 \\ 0 & 0 & 0 & k_{\text{acc}} \end{bmatrix} \qquad A_{\text{dec}} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -k_{\text{dec}} & 0 \\ 0 & 0 & 0 & -k_{\text{dec}} \end{bmatrix}$$

$$A_{\text{right}} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -k_{\text{steer}} \\ 0 & 0 & k_{\text{steer}} & 0 \end{bmatrix} \qquad A_{\text{left}} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & k_{\text{steer}} \\ 0 & 0 & -k_{\text{steer}} & 0 \end{bmatrix}$$

We choose the acceleration parameter $k_{\text{acc}} = 0.5$, the deceleration parameter $k_{\text{dec}} = 2$ and the steering parameter $k_{\text{steer}} = 1$. This results in the corresponding state transition matrices:

$$\Phi_{\text{step}} = \begin{bmatrix} 1.000 & 0.000 & 0.500 & 0.000 \\ 0.000 & 1.000 & 0.000 & 0.500 \\ 0.000 & 0.000 & 1.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 1.000 \end{bmatrix}$$

$$\Phi_{\text{jump}} = \begin{bmatrix} 1.000 & 0.000 & 1.000 & 0.000 \\ 0.000 & 1.000 & 0.000 & 1.000 \\ 0.000 & 0.000 & 1.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 1.000 \end{bmatrix}$$

$$\Phi_{\text{acc}} = \begin{bmatrix} 1.000 & 0.000 & 0.568 & 0.000 \\ 0.000 & 1.000 & 0.000 & 0.568 \\ 0.000 & 0.000 & 1.284 & 0.000 \\ 0.000 & 0.000 & 0.000 & 1.284 \end{bmatrix}$$

$$\Phi_{\text{dec}} = \begin{bmatrix} 1.000 & 0.000 & 0.316 & 0.000 \\ 0.000 & 1.000 & 0.000 & 0.316 \\ 0.000 & 0.000 & 0.368 & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.368 \end{bmatrix}$$

$$\Phi_{\text{right}} = \begin{bmatrix} 1.000 & 0.000 & 0.479 & -0.122 \\ 0.000 & 1.000 & 0.122 & 0.479 \\ 0.000 & 0.000 & 0.878 & -0.479 \\ 0.000 & 0.000 & 0.480 & 0.878 \end{bmatrix}$$

$$\Phi_{\text{left}} = \begin{bmatrix} 1.000 & 0.000 & 0.479 & 0.122 \\ 0.000 & 1.000 & -0.122 & 0.479 \\ 0.000 & 0.000 & 0.878 & 0.479 \\ 0.000 & 0.000 & -0.480 & 0.878 \end{bmatrix}$$

The resulting set of actions that defines the domain model is summarized in Table 1.

The velocity is only transformed linearly (scaled up by acceleration, scaled down by deceleration, rotated by steering) by the actions of the planning domain and therefore $\Psi_a = \mathbf{0}$ for all actions a. Thus, the initial state has to have an initial velocity, and the velocity vector determines the robot's viewing direction.

Despite the fact that the underlying dynamic system is simplified, it inherently captures issues of the real world's physics. For instance, the radius when steering to the right

Name	Precondition P	Numeric Effect E_n	Logic Effect E_l
`step`	$\top$	$\Phi_{\text{step}}\,\mathbf{s} + \Psi_{\text{step}}$	$\neg$`right` $\wedge\neg$`left`
`accelerate`	$\top$	$\Phi_{\text{acc}}\,\mathbf{s} + \Psi_{\text{acc}}$	$\neg$`right` $\wedge\neg$`left`
`decelerate`	$\top$	$\Phi_{\text{dec}}\,\mathbf{s} + \Psi_{\text{dec}}$	$\neg$`right` $\wedge\neg$`left`
`jump`	$v_x^2 + v_y^2 > 4$	$\Phi_{\text{jump}}\,\mathbf{s} + \Psi_{\text{jump}}$	$\neg$`right` $\wedge\neg$`left`
`steer right`	$\neg$`left`	$\Phi_{\text{right}}\,\mathbf{s} + \Psi_{\text{right}}$	`right`
`steer left`	$\neg$`right`	$\Phi_{\text{left}}\,\mathbf{s} + \Psi_{\text{left}}$	`left`

Table 1: Action list of the `Jumpbot` domain.

or to the left hand side depends on the velocity of the system. Also the step size varies with the velocity of the robot. We only check the modeled constraints at discrete steps and neglect the continuous evolution of the system in the meantime, since jumping across the restricted areas in the state space is explicitly allowed. In fact, the domain model may corresponds to an over-simplified representation of a real-world walking robot, but it serves as an interesting example for a system with switched dynamics.

Benchmark Instances

In this section we illustrate the 20 benchmark instances used in the `Jumpbot` domain. We give insights to the difficulties which arise in the instances and visualize the scenarios in order to make it more apparent where planners might fail.

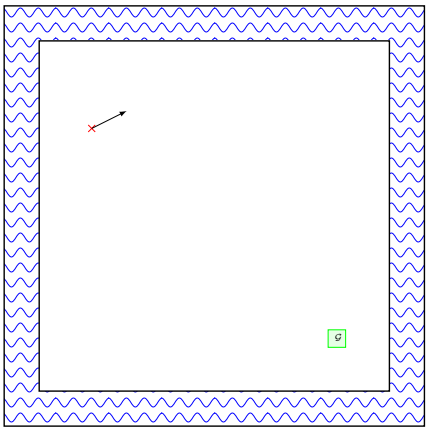

Figure 2: Scenario “plain” (1)

The first scenario is depicted in Figure 2. The “plain”-instance has no obstacles, but the planner has to `steer right` in order to move towards the goal. While it is intended to be a straightforward instance, some planners might find it difficult that due to the lack of obstacles, the search space is not as constrained as in the other problems.

The instances from Figure 3 introduce narrow water obstacles. The ditches are near enough to step over unless the speed of the robot is extremely low. Instances 3 (“two ditches”) and 4 (“bent ditches”) make it more difficult to step over the water because there are more constraints on feasible step positions. This encourages planners to come up with a plan where the robot `jumps` over at least one of the ditches.

Figure 4 introduces a set of instances which have a “moat”, a water obstacle which is too wide to step over so that the planner has to execute a `jump` action. Scenario 5 offers enough space for acceleration and deceleration so that a greedy approach can be successful. In the sixth instance, the robot is looking away from the goal and about to fall into the water. Still, if it manages to do a big left turn, it will most likely have taken enough run to jump over the moat. A problem is then to manage the remaining deceleration space. This problem is taken to an extreme in the seventh scenario “moat local minimum”: here the robot has to steer away from the goal in order to build up enough momentum to jump over the moat. This scenario misleads relaxation based heuristics, because the initial velocity is near 0, the x-position is already correct, and without the moat, the position in direction y is not that far from the goal. The robot is stuck in a local minimum for relaxation heuristics.

The instances from Figure 5 play with the modeling possibilities of the `Jumpbot` domain. Scenario 8 is a chessboard, which makes the planner to rather walk along diagonals than to `jump` around. Scenario “snake” (9) is a snake which offers either the possibility to follow the long path without jumping or to take a shortcut by jumping over the loop. The “smiley” scenario 10 shows that also round obstacles can be modeled. The mouth is a convex obstacle where jumping is prohibited by the lack of space for deceleration.

Figures 6 and 7 depicts the instances used as benchmarks in the dissertation of Löhr (2014). The instances vary only in the initial pose of the robot and the location of the goal zone.

Two scenarios with a “broken bridge” are shown in Figure 8. The robot has to jump over a wide water obstacle, and the feasible jump positions are very restricted. While the initial pose in the seventeenth scenario already guides the planner towards the goal, it is harder to find the correct jump position in scenario 18.

The benchmark set concludes with two hard “wharf” instances depicted in Figure 9.

`PDDL` Encoding

In this section we explain the structure of the `PDDL` files of the `Jumpbot` domain. As we cannot provide a tool to generate instances, this section intends to give enough understanding of the domain to come up with new instances any the less. As a running example we use the `PDDL` definition of the island scenarios 11 to 16.

The modeling of water obstacles is done using a derived predicate `crashed`, a logic formula over comparisons of

arbitrary numeric expressions. For a more complex example, we advise to look at the source code of Scenario "smiley" (10). Some planners flatten nested logic formulas which can result in an exponential blowup. Therefore, we advise domain creators to introduce auxiliary predicates to simplify complex water obstacles. For example consider the the definition of the `crashed` predicate in Figure 10. The island size is restricted to 10 x 10 modeled by the first four inequalities in the big disjunction (Lines 48–51). The "lake" on the lower island is modeled in Line 52 and the "corner" is "removed" from the lake in Line 53. Finally, The robot may also not enter the area between y-coordinate 4 and 8 (Line 54), except the two salients (Lines 55–56) and the two islands (Lines 57–58) which are modeled by auxiliary predicates.

The actions are modeled as shown by means of the `step` example in Figure 11: The only precondition of the `step` action is that the robot is not crashed (Line 62). The numeric effect is written in a form which makes the originating matrix

$$\Phi_{\text{step}} = \begin{bmatrix} 1.000 & 0.000 & 0.500 & 0.000 \\ 0.000 & 1.000 & 0.000 & 0.500 \\ 0.000 & 0.000 & 1.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 1.000 \end{bmatrix}$$

(the first four factors) and vector $\Psi = (0, 0, 0, 0)^\intercal$ (the last row) obvious (Lines 65–68). Lines 69–70 cover the propositional effects while Line 71 increases the special fluent `total-cost`.

A sample problem file is shown in Figure 12. The variables are initialized to the numeric values which are depicted in the scenario descriptions of the previous section. The goal region is a simple conjunction again which defines a square with a length of 0.5 and the velocity has to be close to 0 too. By altering the initial state and the goal location, new instances can easily be generated while fixing the obstacles from the domain file. Attention has to be paid that the velocity vector (at least one of the velocities) is different from zero.

A zip-file containin the 20 benchmark instances of the `Jumpbot` domain can be found online here: `http://gki.informatik.uni-freiburg.de/papers/aldinger-loehr-tr279.zip`

References

[2003] Fox, M., and Long, D. 2003. `PDDL2.1` : An Extension to `PDDL` for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research 20 (JAIR 2003)* 61–124.

[2014] Löhr, J. 2014. *Planning in Hybrid Domains: Domain Predictive Control*. Ph.D. Dissertation, University of Freiburg, Germany.

Appendix

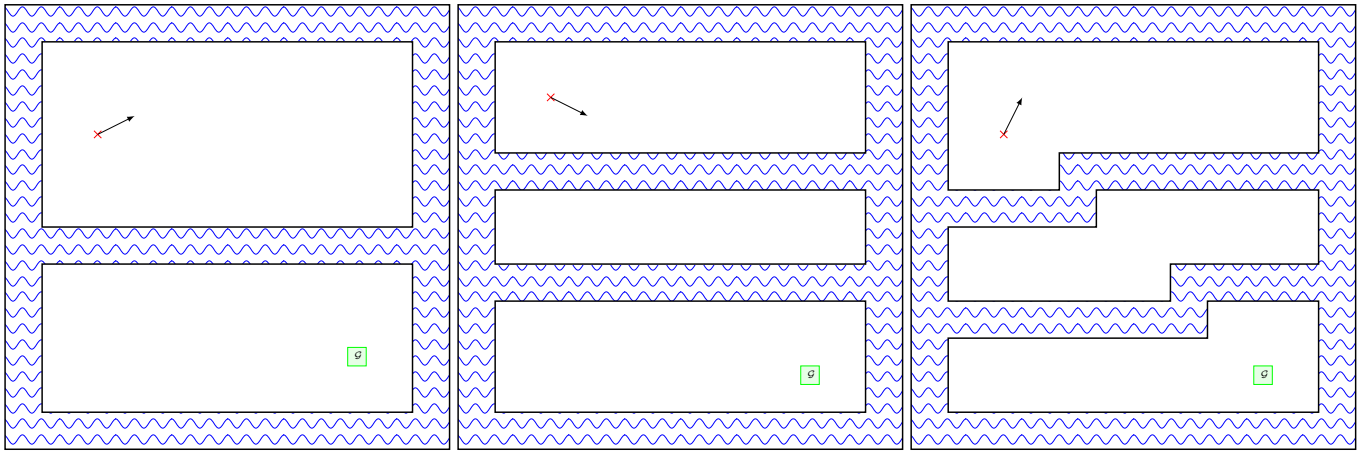

Figure 3: Scenario “ditch” (2), “two ditches” (3) and “bent ditches” (4)

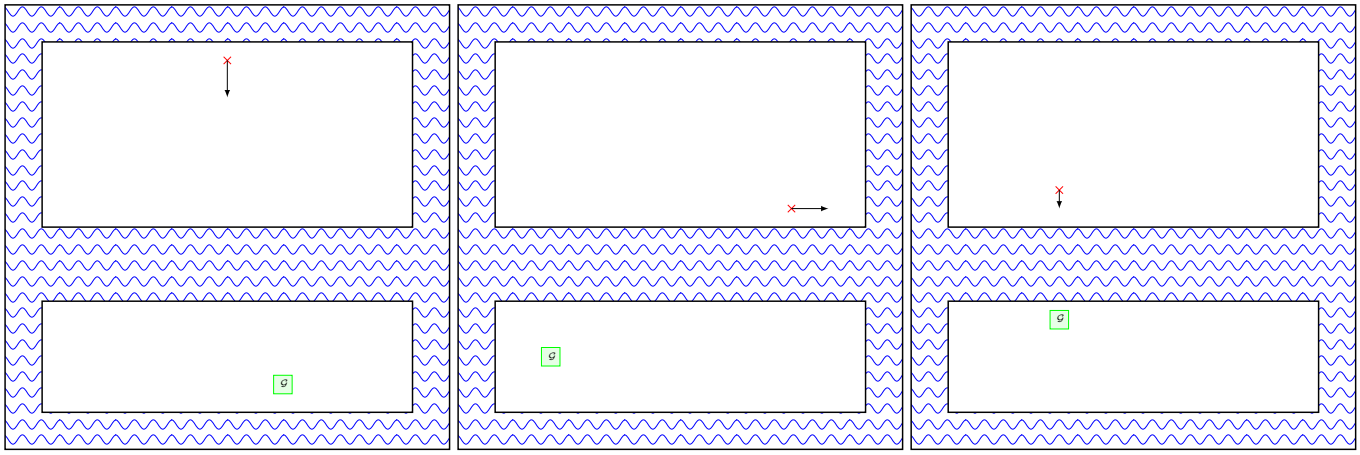

Figure 4: Scenario “moat” (5), “moat wrong viewing direction” (6) and “moat local minimum” (7)

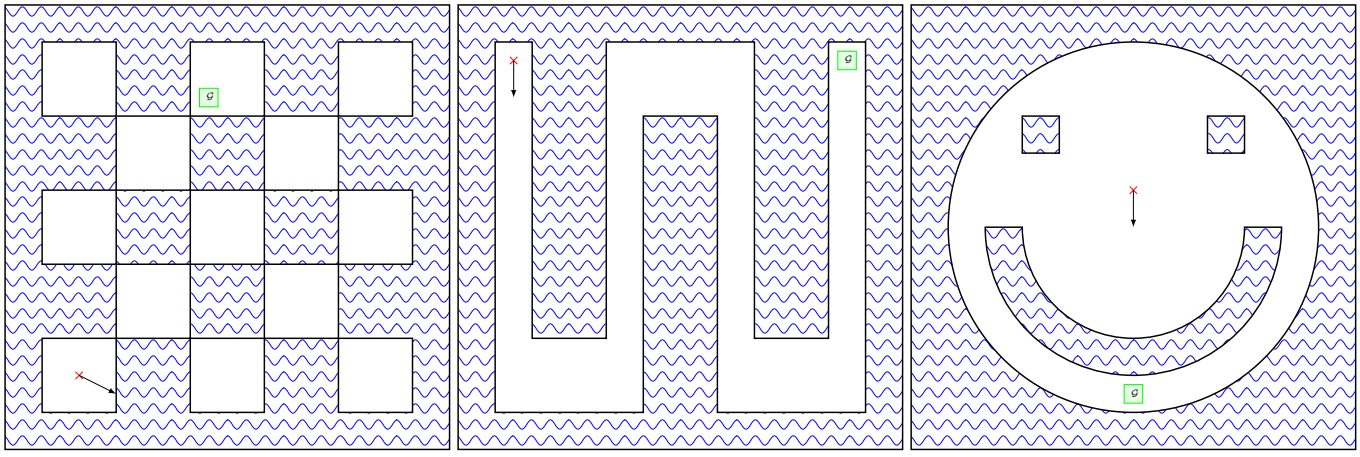

Figure 5: Scenario “chessboard” (8), “snake” (9) and “smiley” (10)

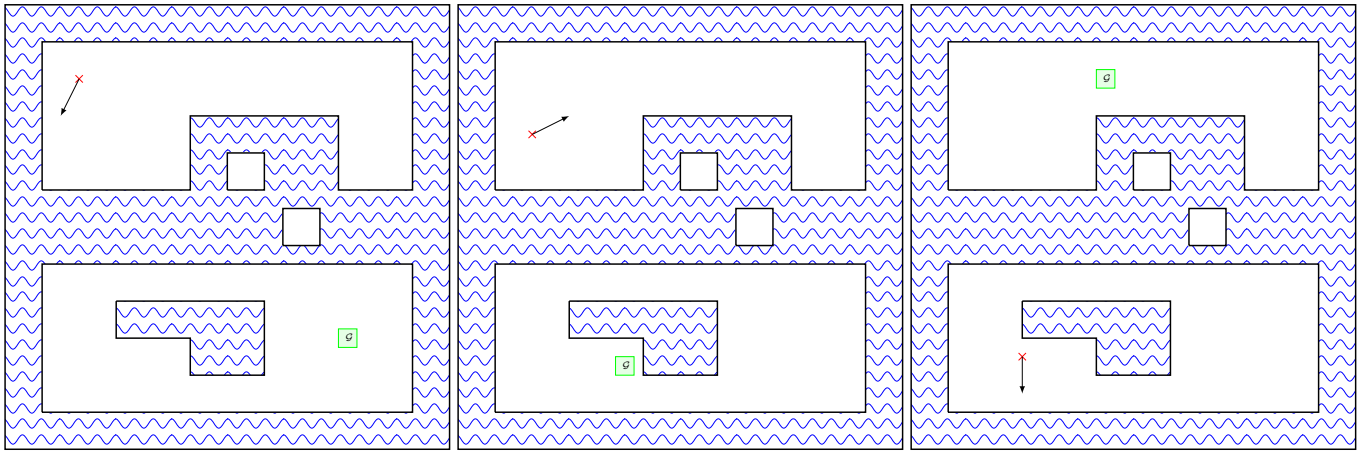

Figure 6: Scenario “islands 1” (11), “islands 2” (12) and “islands 3” (13)

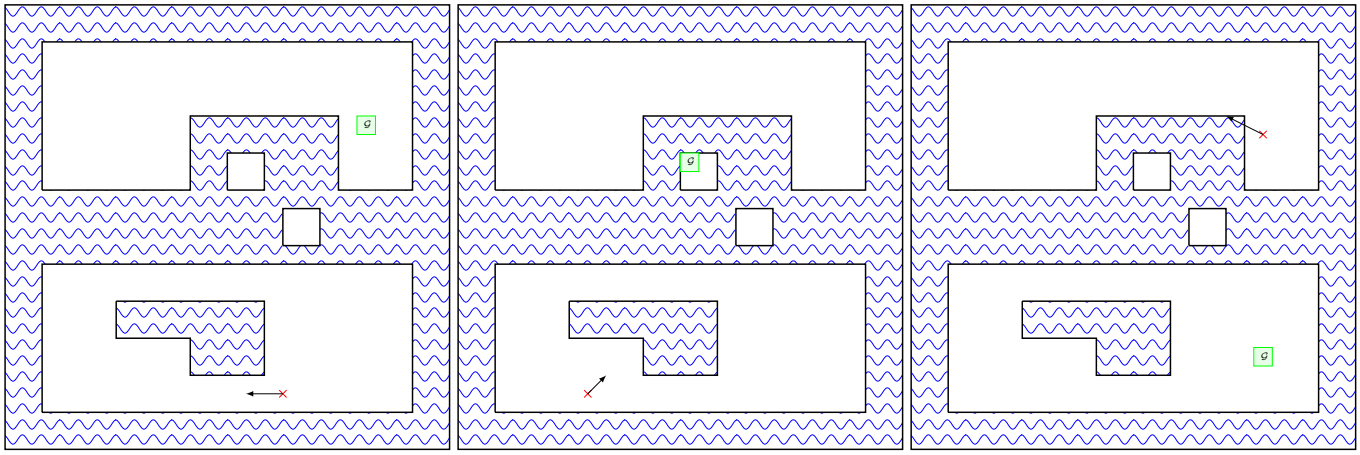

Figure 7: Scenario “islands 4” (14), “islands 5” (15) and “islands 6” (16)

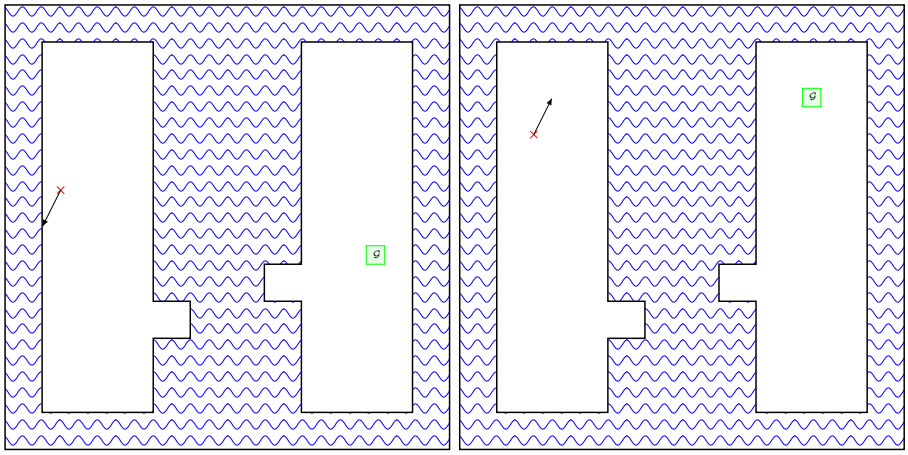

Figure 8: Scenario “broken bridge easy” (17) and “broken bridge” (18)

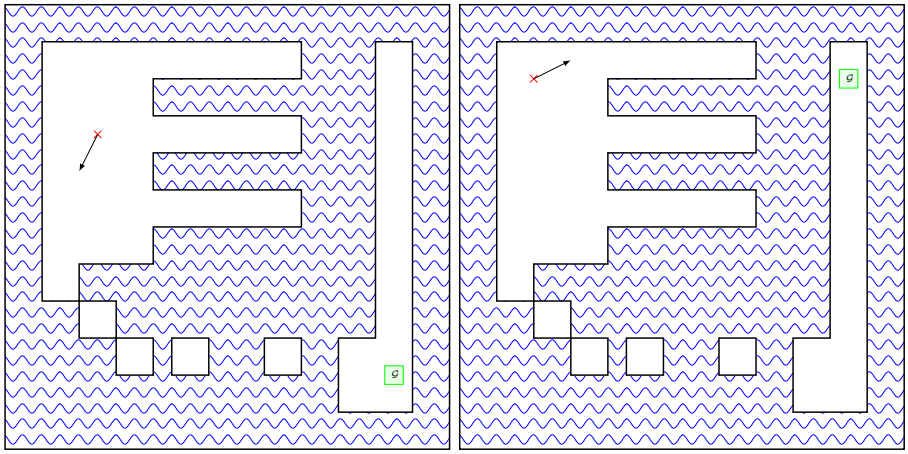

Figure 9: Scenario “around the wharf easy” (19) and “around the wharf” (20)

```
(:derived (crashed)
   (or   (< x 0)                ;; the first four disjunct terms
         (> x 10)               ;; model the 10 x 10 island
         (< y 0)
         (> y 10)
         (and (> x 2) (< x 6) (> y 1) (< y 3) ;; the "lake" at the lower island
              (or (> x 4) (> y 2)))
         (and (> y 4) (< y 8)       ;; the water area between the islands
              (crashed-helper1)     ;; the left salient
              (crashed-helper2)     ;; the right salient
              (crashed-helper3)     ;; the upper left tiny island
              (crashed-helper4)))) ;; the lower right tiny island
```

Figure 10: `PDDL` code for the `crashed` predicate

```
 (:action step
  :parameters ()
  :precondition  (not (crashed))
  :effect
      (and
          (assign x  (+ (* 1 x) (+ (* 0 y) (+ (* 0.5 vx) (+ (* 0   vy) 0)))))
          (assign y  (+ (* 0 x) (+ (* 1 y) (+ (* 0   vx) (+ (* 0.5 vy) 0)))))
          (assign vx (+ (* 0 x) (+ (* 0 y) (+ (* 1   vx) (+ (* 0   vy) 0)))))
          (assign vy (+ (* 0 x) (+ (* 0 y) (+ (* 0   vx) (+ (* 1   vy) 0)))))
          (not (left))
          (not (right))
          (increase (total-cost) 0.5)))
```

Figure 11: `PDDL` code for the `step` action

```
(define (problem p11-islands1)
(:domain jump-bot)

(:init
  (= (x) 1)
  (= (y) 9)
  (= (vx) -0.5)
  (= (vy) -1)
  (not (right))
  (not (left)))

 (:goal
   (and
    (>= (x) 8)
    (<= (x) 8.5)
    (>= (y) 1.75)
    (<= (y) 2.25)
    (>= (vx) -0.2)
    (<= (vx) 0.2)
    (>= (vy) -0.2)
    (<= (vy) 0.2)))

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

Figure 12: PDDL code of a problem file