Snake Domain for HTN IPC 2020

Maurício Cecílio Magnaguagno
Independent researcher
maumagnaguagno@gmail.com

Abstract

This is a description of the Snake domain and problem generator submitted to the HTN IPC 2020 total order track. In the Snake domain the goal is to hunt mice spread over multiple locations, with one or more snakes that get longer as they strike each mouse.

Introduction

The Snake domain is based on the homonymous game genre, in which snakes move to clear locations or strike a nearby mice in a grid/graph-based scenario, the mice do not move as they are too afraid. Each snake occupies one or more adjacent locations due to their long body. The goal is to hunt all the mice or have the snakes occupying certain locations (which forces them to eat and grow). Multiple plans may exist in some scenarios due to snakes being able to strike mice with different orderings and paths. Plans contain zero or more movement actions and one strike per mouse. Differently from the game where usually only one mouse is visible at a time, all mice are visible to give more choice. The domain was motivated by the creative way in which one can describe the snake actions without updating all the snake parts and the little amount of objects required to describe a snake. This paper presents the Snake domain and problem generator[1] for PDDL (McDermott et al. 1998), HDDL (Höller et al. 2020) and (J)SHOP input language (Ilghami and Nau 2003).

Domain

The domain requires *:typing*, *:equality* and *:negative-preconditions* in PDDL, and also *:method-preconditions* and *:universal-preconditions* in HDDL. The JSHOP domain implicitly has the same HDDL requirements. Universal preconditions are used to verify that every location does not contain a mouse and the hunting task is complete.

Types

All objects are either *snake* or *location*. This removes the need to have more objects to define each mouse and snake parts. Removing such objects makes descriptions simpler and grounding faster due to fewer parameters. We use *(mouse-at ?location)* instead of *(at ?mouse ?location)* to remove the *?mouse* parameter from the *strike* action. If we had opted for snake parts we would have multiple descriptions of each long snake, causing a state-space explosion.

Predicates

The state is described by only a few predicates. Locations are *occupied* to avoid overlapping snake parts and mice during movement actions, and also used to simulate walls. Locations that are *adjacent* constrain the range of actions. A snake *head* location is used to constrain the range of actions of each snake. The sequence of locations occupied by each snake are *connected*, with the last part being the *tail*.

Actions/Operators

Three actions/operators exist in this domain. The *strike* action represents the mouse being consumed by an adjacent snake head. Two movement actions are used to describe a single or multiple location snake movement, *move-short* and *move-long*, respectively. Move was split in two to minimize the amount of ground actions without the use of disjunctions. The JSHOP version also contains explicit *visit/unvisit* operators to avoid infinite loops. The signatures of actions are shown in Listing 1.

Listing 1: Signatures of *Snake* actions with types ommited.

```
(:action strike :parameters (
  ?snake ?headpos ?foodpos))
(:action move-short :parameters (
  ?snake ?nextpos ?snakepos))
(:action move-long :parameters (
?snake ?nextpos ?headpos ?bodypos ?tailpos))
```

Tasks and Methods

Two tasks are described in the JSHOP and HDDL versions, with 5 methods in total. The first task is *hunt*, with zero parameters, used as the main task. Two methods are used for this task, a recursive one to select one snake that will strike a mouse, and a base one for no more mice. The base case is described after the recursive method as it happens only once, when all mice have been consumed.

The second task is *move*, with a snake, its head and goal location as parameters. Here we have a base method and two recursive ones to use the *move-long* and *move-short* actions.

[1] https://github.com/Maumagnaguagno/Snake

The *move-base* case is described first to avoid redundant expansions in planners that follow the description order. The *move-short* is the last case described as it is less common. The signatures of tasks and their related methods are shown in Listing 2.

Listing 2: Signatures of *Snake* tasks and related methods.

```
(:task hunt :parameters ())
(:method hunt_all :parameters (?snake
  ?foodpos ?snakepos ?pos1))
(:method hunt_done :parameters ())
(:task move :parameters (?snake
  ?snakepos ?goalpos))
(:method move-base :parameters (
  ?snake ?snakepos ?goalpos))
(:method move-long-snake :parameters (
  ?snake ?snakepos ?goalpos ?pos2
  ?bodypos ?tailpos))
(:method move-short-snake :parameters (
  ?snake ?snakepos ?goalpos ?pos2))
```

Problem

Each problem contains snakes and locations as objects. Each snake must contain at least a head and tail described in the initial state. If head and tail are on the same location, single location snake, there is no need to connect snake parts. Each mouse location must be described in the initial state. Locations that contain snake parts, mice or walls are occupied. Locations must be adjacent to one another to describe possible paths. Adjacencies are usually symmetrical, *(adjacent l1 l2) (adjacent l2 l1)*, and grid-based, but are not limited to.

For goal-based planning it may include snakes' final configuration and mice not existing anymore. For task-based planning it may include movement and hunting tasks. Due to the possibly large amount of mice, it is recommended to use a quantifier to describe a goal state without mice or tasks to hunt every mouse.

Problem generator

Currently a text representation, like the one from Sokoban[2], can be used with our problem generator. Each character in a text file represents one element of the Snake problem in a grid-based scenario:

- *Space*: clear location
- @: snake head location
- $: snake body location
- *: mouse location
- #: wall location

Currently limited to a single snake with snake parts adjacent only to previous and next locations to avoid ambiguity. Walls are converted to always occupied locations, but could also be represented as lack of adjacencies to these locations, which would be harder to manually modify later. Multiple problems in this format are already available, they were manually crafted to generate longer solutions or force certain paths for the snake to be able to strike all mice.

The current problem generator converts all **.snake* files in the current folder or the ones provided as arguments according to a *type* argument, generating **.snake.type* files. Type includes *pddl*, *hddl* and *jshop*.

[2]http://www.sokobano.de/wiki/index.php?title=Level_format

Example

The content of the input *pb2.snake* is presented in Listing 3. With the execution of the problem generator, *ruby pbgenerator.rb hddl pb2.snake*, we obtain an HDDL equivalent problem. The output *pb2.snake.hddl* is presented in Listing 4.

```
*␣␣\n
␣␣$\n
␣␣@
```

Listing 3: Snake input file example with 3x3 grid, two-parts snake and a mouse.

Listing 4: HDDL description of converted *pb2.snake*.

```
(define (problem pb2) (:domain snake)
  (:objects viper - snake
            px0y0 px1y0 px2y0
            px0y1 px1y1 px2y1
            px0y2 px1y2 px2y2 - location)
  (:init (head viper px2y2)
         (connected viper px2y2 px2y1)
         (tail viper px2y1)
         (mouse-at px0y0)
         (occupied px0y0)
         (occupied px2y1)
         (occupied px2y2)
         (adjacent px0y0 px1y0)
         ... ; Adjacencies ommited
         (adjacent px2y2 px2y1))
  (:htn :subtasks (hunt)) )
```

Conclusion

This domain presents several features to help planner testing. All planning instances can be described in the compact format used by the generator, converted to images and easily modified by hand. The planning instances can scale indefinitely, as larger grids accept more mice and longer snakes, however it requires a smart random level generator to create such larger instances with unique challenges. Heuristic planners can estimate which snake is closer to each mouse to minimize actions, while considering that long snakes create moving walls that affect such estimations. Numeric planners could take even more advantage in regular grids. In the future we expect to improve the problem generator with multiple snakes and their goal locations. Multiple snakes could also modify the domain, with the requirement of moving all snakes every time-step, like the real game.

References

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020)*, 9883–9891. AAAI Press.

Ilghami, O., and Nau, D. S. 2003. A General Approach to Synthesize Problem-Specific Planners. Technical Report CS-TR-4597, Maryland University, Dept of Computer Science, College Park, Maryland.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL-the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.