Probabilistic Programs as an Action Description Language

Ronen I. Brafman, David Tolpin and Or Wertheim

Department of Com Ben Gurion University of the Negev
{brafman,tolpin,orwert}@post.bgu.ac.il

Abstract

Actions description languages (ADLs), such as STRIPS, PDDL, and RDDL specify the input format for planning algorithms. Unfortunately, their syntax is foreign to most potential users of planning technology. Moreover, this syntax limits the ability to describe complex and large domains. We argue that programming languages (PLs), and more specifically, probabilistic programming languages (PPLs), provide a more suitable alternative. PLs are familiar to all programmers, support complex data types and rich libraries for their manipulation, and have powerful constructs, such as loops, sub-routines, and local variables with which complex, realistic models and complex objectives can be simply and naturally specified. PPLs, specifically, make it easy to specify distributions, which are essential for stochastic models. The natural objection to this proposal is that PLs are opaque and too expressive, making reasoning about them difficult. However, PPLs also come with efficient inference algorithms, which, coupled with a growing body of work on sampling-based and gradient-based planning, imply that planning and execution monitoring can be carried out efficiently in practice. We expand on this proposal, illustrating its potential with examples.

Introduction

Action description languages (ADLs), such as STRIPS (?), PDDL (?) and RDDL (?) specify the input format to planning algorithms. Unfortunately, their syntax is familiar to planning experts only, and not to potential users of planning technology. This has been recognized as a detriment to wider adoption of planning technology, and has led to much interest in the topic of knowledge acquisition for planning with an annual workshop.

The syntax of PDDL or RDDL can be mastered with some practice, but it comes with inherent expressive limitations. These make the task of modeling large and complex real-world planning problems particularly challenging. In this paper we argue for the use of programming languages (PLs), and more specifically, probabilistic programming languages (PPLs) (????), as more suitable alternatives to classical ADLs.

First, programmers know quite well how to use PLs to describe what they conceive in their minds. Second, it is easier to write complex models in PLs because they support complex data types and come with rich and efficient libraries for their manipulation. Describing arrays, linked-lists, vectors, trees, tensors etc. is very difficult and cumbersome within a classic ADL. Such a description would not be transparent to anyone who reads it, is likely to be lengthy, and algorithms that use it would have no knowledge of its special properties. PLs also come with powerful constructs, such as loops, sub-routines, and local variables that farther enhance our ability to naturally specify complex, realistic models and complex objectives. More specifically, PPLs make it easy to specify distributions essential for modeling uncertainty about action effects and about the world's state — essential components of many, if not most, real-world models. Moreover, most ADLs are restricted to closed models, whereas open models are easy to specify using PPLs.

Two related objections arise at this point: First, isn't this essentially telling us to use simulators? Second, we know from KR that expressive models are difficult to reason with, and more specifically, in the case of full-fledged PLs, many inference tasks would be undecideable. Moreover, PLs are likely not structured enough to support good algorithms.

Indeed, simulation will play a major role in planning algorithms based on PPL models. Simulation based methods such as algorithms for bandit problems (?), MDPs (?) and POMDPs (??), RL algorithms (?), and Novelty-based methods (?) have been very successful, recently. And by using code to describe the model, we make it very easy to automatically generate efficient simulators (?) that can also utilize various libraries for manipulating complex data types.

But PPL-based ADLs offer more than fast simulation. We believe that future work can exploit the growing body of supported inference algorithms in service of planning and (no less important) execution monitoring. Specifically, PPLs make it easy to assess the likelihood of new observations, allowing us to detect rare and unlikely events. They support more involved inference algorithms that can allow us to validate our plans and verify various properties. PPLs also support annotation methods that are then used by the inference algorithms to make the computation more efficient. One can use such annotations to construct more structured abstractions of the model that can guide computation (e.g., bias the sampling process towards better actions, provide heuristic estimates), or to automatically detect components with special structure (e.g., deterministic variables) and exploit them.

Perhaps more importantly, PPLs can exploit automatic differentiation methods (?). This implies that recent efficient gradient-based planning methods, such as (?) could be used to solve them.

We believe the above makes the case for exploiting PPLs-based ADL in planning clear. The rest of this paper provides some background, expands on some of the above issues, and describes a number of examples that illustrate the points made above. We end with a suggested research agenda.

Background

Action Description Languages

The basic components of most planning ADLs can be traced to the STRIPS language (?). Propositions are used to describe the state of the world. These propositions are obtained by instantiating typed variables within a given set of predicates, by objects. For example On(BlockA,BlockB) instantiates the binary predicate On using two block objects, which yields a proposition that is either true or false. Actions are usually specified as schema, e.g., Pickup(?x), with its variables as place holders for appropriate objects. The action description specifies an applicability condition in the form of the precondition list — a list of predicates with objects or variables that appear in the action schema, and a similar list describing its effects.

Planning languages have evolved to gradually more complex descriptions, including quantification over variables, conditional effects, resources, cost, preferences, constraints, axioms, non-Boolean finite-domain variables and much more (???). However, the constrained structure of the languages, their reliance on simple data types and lack of iterators makes it difficult to express complex models.

Moreover, none of the above languages support probabilistic models — an essential component of realistic models. These models require numeric information and, often, variables with continuous domains. While an extension of PDDL to probabilistic domains (PPDDL) (?) exists, it is mainly useful for transforming existing classical domains to probabilistic ones.

There are two notable exceptions. RDDL (?) is a language developed for describing dynamic Bayesian networks (DBNs) (?) that capture the transition and observation functions of MDPs and POMDPs. RDDL can be viewed as restricted, special purpose probabilistic programming language. DBNs describe the conditional probability of post-action or observation values given the pre-action variables' values. Additional layers in between the pre-action and post-action layer allow intermediate computations. RDDL specs are declarative and explicitly specify the conditional probabilities of each variable value. These values can be described using a broad set of supported mathematical and logical functions and expressions. A variable in one layer can be conditioned on the previous layer, only. This can be viewed as a generative model that describes how each layer's values are generated from the previous layer. Writing RDDL specifications requires mastering their syntax, and complex distributions may be difficult to specify. Intermediate computations are accomplished by adding intermediate DBN layers, which may be tedious since each variable can only be assigned once. As we show later, a piece of code with local variables (possibly assigned multiple times) and advanced control structures like loops can be much easier to specify, is more compact and is easier to understand. It also enables much more efficient sampling by simply using the code itself, whereas RDDL specification must be parsed and sampled by a generic piece of code. And finaly, RDDL can describe graphical models, only. Hence, it cannot capture open-world domains.

BLOG (?) was an important step in the direction we are espousing worth highlighting. Its syntax is PL-like, and it supports set objects using which it is able to model open universes. However, it is short of a full-fledged PPL as it does not support rich data structure and itreserators, and lacks the inference capabilities of modern PPLs.

Probabilistic Programming

Probabilistic programming (????) represents statistical models as programs written in an otherwise general programming language that provides syntax for the definition and conditioning of random variables. This means that programmers familiar with languages such as C++, Python, and more, can use them with almost no additional effort. Inference can be performed on probabilistic programs to obtain the posterior distribution or point estimates of the variables. Inference algorithms are provided by the PPL framework, and each algorithm is usually applicable to a wide class of probabilistic programs in a black-box manner. Probabilistic programs may contain loops, conditional statements, recursion, and operate on built-in and user defined data structures. The algorithms include Metropolis-Hastings(??), Hamiltonian Monte Carlo (?), expectation propagation (?), extensions of Sequential Monte Carlo (?????), variational inference (??), gradient-based optimization (??), and others.

Expressive Power

We compare PPL-based specification with RDDL ones on two examples, seeking to illustrate that that PPL-based specification are easier to write and understand

and can describe models that RDDL (and most other ADLs) cannot describe, and this is done without a need to learn a new language. We focus on RDDL because it is the richest available ADL actively used in planning that supports probabilistic models.

Locusts Swarm — Multi-Stage Exogenous Events

In this domain, aside from the agent's action, there are exogenous events that take place in parallel. To capture these exogenous events, RDDL needs to introduce intermediate layers in the DBN. If the event is complex, i.e., develops in multiple stages, then one layer is needed for each stage since every variable in a DBN can be assigned once only. With code, on the other hand, one can considerably simplify the domain description by using intermediate variables that describe the process and are reassigned multiple times.

In the Locusts Swarm domain, a swarm of locusts has invaded a nature reserve. The reserve is divided into nine primary cells that host a variety of endangered plants (10 tons of plants in each cell). The swarm moves from cell to cell twice during the night, eating half the plants of each cell it encounters. It rests in the last cell visited at night during the following day. The swarm moves stochastically depending on how many plants it smells in neighboring cells. The reserve management has a crop duster that can spray pesticides at a single cell at night. These materials fade away after one day and need a whole day to disable the swarm. The only chance to stop this catastrophe is to spray, in advance, the cell in which the swarm will rest the following day.

RDDL requires a description that grows linearly because increasing the number of nightly swarm transitions increases the number of required RDDL layers. Using a PPL, we can provide a fixed-size description. The verified RDDL code appears in Listing 3 and the pseudo PPL code in Listing 1.

The Open World Room Cleaning Domain

Using code, it is easy to describe open-world domains, i.e., domains in which the set of objects is not constant. This is very natural in many cases, and was demonstrated in BLOG (?). Below is a simple example of a domain describing a robot cleaning a children's room. Such a domain does not correspond to a graphical model, because the size of the graph changes. Hence, it cannot be described in RDDL, PDDL, and similar languages.

In the Room Cleaning domain, a robot cleans the children's room and should put all the toys in their place. Each toy has a size and a level of difficulty to grasp, and a price. The robot is initially aware of only three toys, some already in place. It can place one toy per minute, yet it may drop and break a toy. Every minute it receives positive reward for ordered toys based on their size, and a negative reward for unordered toys. If it breaks a toy, it receives a one-time penalty for its price. Every minute the robot is cleaning, it may find up to two new toys needing attention. The mission only ends when all toys are in place, and there is no maximum number of toys (See pseudo code in Listing 2).

Reward Machines and Factorization

Recently, specification of complex, non-Markovian reward functions (i.e., ones depending on the entire past), captured by automata, and known as reward machines have become popular (?). By taking the Cartesian product of the underlying MDP and the reward-machine automaton, we obtain a new, product MDP w.r.t which the reward is Markovian.

Technically, reward machines allow us to decompose the "true" product MDP into smaller components. This is good for learning, representation, and planning. This is particularly important when MDPs are represented using explicit tables. Probabilistic programs generalize this idea. First, factored structure can be reflected directly in the code — both the probabilistic part and the reward part. Second, one can describe much more elaborate reward functions using code. In fact, utilities behave much like log-probabilities (??), and the same machinery can be applied to them.

Inference, Planning and Verification

One advantage of PPLs is the (growing) body of inference algorithms they support. Sampling algorithms are obtained immediately, as the PPL code can be executed to yield simulated values. This is an important advantage over existing ADLs in which the description must be parsed and compiled to an executable format. Below we describe a number of additional inference tasks that can be carried out using existing infra-structure.

Computing Policy Parameters — Sailing Domain

In many applications, we seek to find optimal parameters for a policy with fixed structure. A classic example is a fixed-size finite-state controller, where we seek to find the structure of the best controller for a POMDP with bounded size (??). The Sailing Domain provides an example of such a problem (??).

A sailing boat must travel between the opposite corners A and B of a square lake of a given size. At each step, the boat can head in 8 directions to adjacent squares. It always moves one unit-distance, called a leg. The unit distance cost of movement depends on the wind, which can also blow in 8 directions. The cost of sailing into the wind is prohibitively high, upwind is the highest feasible, and away from the wind is the lowest. When the angle between the boat and the wind changes sign, the sail must be tacked, which incurs an additional cost. The wind is assumed to follow a random walk, either staying the same or switching to an adjacent direction, with a known probability.

For any given lake size, there is a non-parametric stochastic policy that tabulates the distribution of legs for each combination of location, tack, and wind. However, such policy does not generalize well — if the lake surface area increases, due to a particularly rainy year, for example, the policy is not applicable to the new parts of the lake. Instead, we can define a generalizable parametric policy balancing between hesitation in anticipation for a better wind and rushing to the goal at any cost. The policy chooses a leg with log-probability equal, up to a constant, to the sum of the leg cost and of the Euclidean distance between the position after the leg and the goal, multiplied by the policy parameter θ (the leg directed into the wind is excluded from choices). The greater the θ, the higher is the probability that a leg bringing the boat closer to the goal will be chosen:

$$\log \Pr(\mathit{leg}) = \mathit{leg\text{-}cost} + \theta \cdot \mathit{distance}(\mathit{next\text{-}location}, \mathit{goal}) + C \tag{1}$$

In a probabilistic programming language, (1) can be expressed as just sampling from a categorical distribution:

$$\mathit{leg} \sim \text{Categorical}(\{\mathit{leg}_i : \Pr(\mathit{leg}_i)\}) \tag{2}$$

Inferring the policy, either offline or online, can be accomplished using out-of-the-box inference algorithms for models expressed by probabilistic programs. Note, that this policy is differentiable with respect to θ. This allows to leverage automatic differentiation capabilities of modern probabilistic programming languages for efficient optimization.

Verification

Probabilistic model-checking (a.k.a. statistical model checking in the verification community) (?) refers to the problem of verifying certain properties of probabilistic transition systems. Special purpose systems for this task such as PRISM (?) and UPPAAL-SMC (?) have been developed. The basic query in such systems is what is the probability of a future event satisfying some property. There are several settings in which the query is posed: (a) for the fixed policy — to check the robustness of a chosen policy, (b) for a broad class (a distribution) of policies — to check the robustness of the agent in an average case, and (c) for the worst-case policy with respect to the query — to check robustness of the agent in an adversarial setting. For the above mentioned sailing domain, for example, we may query about the probability of visiting the complementary corners of the square lake, where bulrush grows, or about the probability of travel cost execeeding a certain threshold.

This is a standard query in probabilistic programming. From the probabilistic inference point of view, these three settings involve analysis of the predictive posterior of the model and differ only in the way the model is conditioned. For (a), the predictive posterior is obtained for the model conditioned on the chosen policy parameters. For (b), a Bayesian prior reflecting assumptions on the class of policies is imposed on latent variables, and the predictive posterior is obtained under the prior. For (c), applicable to stochastic domains only, the model is conditioned on the occurence of the event of interest. The posterior is most often represented by samples, allowing for convenient and efficient calculation of quantiles and compatibility intervals; alternatively, variational inference may allow performing probabilistic queries on the posterior in closed form.

Moreover, unlike purpose-built tools, which are often confined to a particular type of queries, the whole arsenal of Bayesian model checking, evaluation, and comparison (?, Chapters 6–7) becomes available when probabilistic programming is used to define the model.

Research Agenda

The discussion above suggests natural research questions:

Planning Algorithms The most important direction is continuing to develop planning algorithms that can exploit PPL-based action descriptions. This includes:

- Efficient sampling algorithms.
- Model-based gradient descent algorithms that exploit our ability to auto-diff probabilistic programs.
- Techniques for guiding planners, including heuristics (such as novelty (?)), helpful-actions, as used by POMCP (?) and Despot (?) to perform roll-outs, and other techniques.
- Algorithms that recognize domains with special properties, e.g., deterministic, fully observable, allowing the use of special-purpose planners
- Algorithms that exploit component structure within a program for planning, such as deterministic components (as in reward machines and POMDP-Lite (?)), fully observable components, etc.

Learning Algorithms

A long line of work, going back at least to (?) is concerned with learning action models for planning. This becomes even more crucial when we allow more complex descriptions. This should be of interest to the PPL community, interested in learning PPLs in general, as it offers a more constrained setting with many potential examples. If fact, learning quantitative information with program structure fixed is already supported by existing PPLs.

Compilation

Methods for compiling programs into existing formalisms (e.g., to RDDL) could be used to exploit the strength of current planning algorithms. Approximate compilation methods can be used to provide approximate solutions, and these could also be used to guide algorithms that solve the original problem description, e.g., as heuristics.

Summary

In this paper we argued for the use of PPLs as a language for specifying stochastic planning models. PPL makes it easier to express complex models, and their code can be exploited by planning algorithms. Indeed, models of various realistic systems cannot be captured compactly using existing formalism, nor do associated methods scale to handle planning in them. Moreover, PPL inference algorithms can be used by planning algorithms to support various useful inference tasks. This, of course, does not detract from the usefulness of existing methods with their associated planning algorithms. They remain useful as potential models, when appropriate, and as tools that can be exploited by richer models to support efficient planning.

Appendix: Code Examples

In the following pages we provide PPL pseudo-code for the Locust Swarm and Room Cleaning domains and verified RDDL code for the Locust Swarm domain.

Listing 1 The Locusts Swarm domain in pseudo PPL code.

```
coordinates enum {x,y}
directions enum {up,down,left,right}
overnight_transitions=2 //The number of nightly swarm movements.

//state variables:
tons_of_plants_in_cells[3,3]=10 //10 tons in each of 3x3 cells
swarm_location = (2,2) //The (x,y) location of the swarm.
swarm_disabled=false //Determine if the swarm is disabled and no longer damages the plants.

//'disable_swarm_action' is the location we want to spray the pesticides. 's' is the state we want to sample from.
function sample_next_state(disable_swarm_action,s)
    for move in 1 to overnight_transitions //Repeat by the number of the swarm nightly transitions.
        total_weight=0
        weights[4]

        //Calculate the probability for the swarm to go in each direction based on the amounts of plants there.
        weights[up]= if s.swarm_location[y] > 2 then 0.0 else s.tons_of_plants_in_cells[s.swarm_location[x],s.swarm_location[y]+1]
        weights[down]= if s.swarm_location[y] < 2 then 0.0 else s.tons_of_plants_in_cells[s.swarm_location[x],s.swarm_location[y]-1]
        weights[right]= if s.swarm_location[x] > 2 then 0.0 else s.tons_of_plants_in_cells[s.swarm_location[x]+1],s.swarm_location[y]]
        weights[left]= if s.swarm_location[x] < 2 then 0.0 else s.tons_of_plants_in_cells[s.swarm_location[x]-1],s.swarm_location[y]]
        total_weight = weights[up] + weights[down] + weights[right] + weights[left]

        for move_direction in directions //Normalize the weights.
            weights[move_direction]=weights[move_direction]/total_weight
        move_direction = sample_discrete(weights) //Sample the swarm movement direction.

        //Calculate the new swarm location based on the sampled direction.
        s.swarm_location[x] = switch(move_direction){
                                    case right: s.swarm_location[x]+1
                                    case left: s.swarm_location[x]-1
                                    default: s.swarm_location[x]
                                    }
        s.swarm_location[y] = switch(move_direction){
                                    case up: s.swarm_location[y]+1
                                    case down: s.swarm_location[y]-1
                                    default: s.swarm_location[y]
                                    }
        if not s.swarm_disabled //If the swarm is not disabled, it eats half the food in its current location.
            s.tons_of_plants_in_cells[swarm_location] = 0.5 * s.tons_of_plants_in_cells[swarm_location]

    if s.disable_swarm_action == s.swarm_location //The swarm is disabled if the crop duster sprayed its last location.
        s.swarm_disabled=true

    //The reward is the total number of plants left in the nature reserve.
    reward =0
    foreach tons_in_cell in s.tons_of_plants_in_cells
        reward= reward+tons_in_cell
    return reward,s //Return the reward and the sampled next state.
```

Listing 2 The Room Cleaning domain in pseudo PPL code.

```
toys_properties enum {size, inplace, price, grasping_difficulty}
toys=[{0.1,true,0.3,0.1},{0.8,false,0.2,0.1},{6.8,false,1.5,0.2}] \\The 'toys' list is the only state variable.

//'order_toy_action' is the index of the toy that should be ordered. 's' is the state we want to sample (the next state and reward)
      from.
function sample_next_state(order_toy_action,s)
    reward = 0
    toy = s.toys[order_toy_action] //Get the toy the robot should handle.
    if not toy[inplace] //If toy not in place
        if sample_bernoulli(toy[grasping_difficulty]) //Sample if the toy falls and breaks based on its grasping difficulty.
            reward = -toy[price] //Give a negative reward based on the toy price.
            s.toys.remove(toy) //Remove the broken toy from the toys list.
        else
            toy[inplace]=true //If the robot could grasp the toy, it is said to be in place.

    foreach toy in s.toys //Give a positive (negative) reward for each toy (not) in place by its size.
        if toy[inplace]
            reward = reward + toy[size]
        else
            reward = reward - toy[size]

    number_of_toys_to_add = sample_discrete(0.4,0.3,0.3) //Sample the number of new toys found (zero:0.4, one:0.3, or two:0.3).
    for 1 to number_of_toys_to_add //Add toys as the amount found.
        new_toy = {absolute_value(sample_normal(1,1), sample_bernoulli(0.2),
                                    sample_uniform(0,4), sample_uniform(0,0.4)} //Sample the new toy properties.
        s.toys.append(new_toy) //Add the new toy to the toys list.
    return reward,s //Return the reward and the sampled next state.
```

Listing 3 The Locusts Swarm domain in RDDL code.

```
// swarm_of_locusts: cells map
//  ______________
//   3|@c1|@c2|@c3|
//   2|@c4|@c5|@c6|
//   1|@c7|@c8|@c9|
//  ---------------
//y/x  1   2   3
domain swarm_of_locusts2 {
types {cell : object; direction : {@up, @down, @left,@right};};

pvariables {
x_location(cell) : {non-fluent, int, default = -1};
y_location(cell) : {non-fluent, int, default = -1};
tons_of_plants(cell) : {state-fluent, real, default = 10.0};
swarm_disabled : {state-fluent, bool, default = false};
swarm_location_x : {state-fluent, int, default = 1}; swarm_location_y : {state-fluent, int, default = 1};
total_neighbors_plants1 : {interm-fluent, real, level = 1};
swarm_move1 : {interm-fluent, direction, level = 2};
swarm_location1_x : { interm-fluent, real, level = 3};
swarm_location1_y : { interm-fluent, real, level = 3};
tons_of_plants1(cell) : { interm-fluent, real, level = 4};
total_neighbors_plants2 : {interm-fluent, real, level = 5};
swarm_move2 : {interm-fluent, direction, level = 6};
swarm_location2_x : { interm-fluent, real, level = 7};
swarm_location2_y : { interm-fluent, real, level = 7};
disable_swarm_act(cell) : { action-fluent, bool, default = 0};};

cpfs{
total_neighbors_plants1 = sum_{?c : cell}[if (((swarm_location_x+1) == x_location(?c) ^ swarm_location_y == y_location(?c)) |
                          ((swarm_location_x - 1) == x_location(?c) ^ swarm_location_y == y_location(?c)) |
                          ((swarm_location_y + 1) == y_location(?c) ^ swarm_location_x == x_location(?c)) |
                          ((swarm_location_y -1) == y_location(?c) ^ swarm_location_x == x_location(?c))) then tons_of_plants(?c) else 0.0];
swarm_move1 = Discrete(direction,
     @up : if(swarm_location_y == 3) then 0.0 else (sum_{?c : cell}[if ((swarm_location_y + 1) == y_location(?c) ^
                                              swarm_location_x == x_location(?c)) then tons_of_plants(?c) else 0.0])/total_neighbors_plants1,
     @down : if(swarm_location_y == 1) then 0.0 else (sum_{?c : cell}[if ((swarm_location_y - 1) == y_location(?c) ^
                                              swarm_location_x == x_location(?c)) then tons_of_plants(?c) else 0.0]/total_neighbors_plants1),
     @left : if(swarm_location_x == 1) then 0.0 else (sum_{?c : cell}[if ((swarm_location_x - 1) == x_location(?c) ^
                                              swarm_location_y == y_location(?c)) then tons_of_plants(?c) else 0.0]/total_neighbors_plants1),
     @right :if(swarm_location_x == 3) then 0.0 else (sum_{?c : cell}[if ((swarm_location_x + 1) == x_location(?c) ^
                                              swarm_location_y == y_location(?c)) then tons_of_plants(?c) else 0.0]/total_neighbors_plants1));
swarm_location1_x = switch(swarm_move1){case @up: swarm_location_x,case @down: swarm_location_x,
                                              case @right: swarm_location_x + 1,case @left: swarm_location_x - 1};
swarm_location1_y = switch(swarm_move1){case @up: swarm_location_y + 1, case @down: swarm_location_y - 1,
                                              case @right: swarm_location_y,case @left: swarm_location_y};
tons_of_plants1(?c)= if (swarm_location1_y == y_location(?c) ^ swarm_location1_x == x_location(?c)) then tons_of_plants(?c)*0.5
                                              else tons_of_plants(?c);
total_neighbors_plants2=sum_{?c : cell}[if (((swarm_location1_x+1) == x_location(?c) ^ swarm_location1_y == y_location(?c)) |
                          ((swarm_location1_x - 1) == x_location(?c) ^ swarm_location1_y == y_location(?c)) |
                          ((swarm_location1_y + 1) == y_location(?c) ^ swarm_location1_x == x_location(?c)) |
                          ((swarm_location1_y -1) == y_location(?c) ^ swarm_location1_x == x_location(?c))) then tons_of_plants1(?c) else
                                0.0];

swarm_move2 = Discrete(direction,
     @up : if(swarm_location1_y == 3) then 0.0 else (sum_{?c : cell}[if ((swarm_location1_y + 1) == y_location(?c) ^
                                              swarm_location1_x == x_location(?c)) then tons_of_plants1(?c) else 0.0])/total_neighbors_plants2,
     @down : if(swarm_location1_y == 1) then 0.0 else (sum_{?c : cell}[if ((swarm_location1_y - 1) == y_location(?c) ^
                                              swarm_location1_x == x_location(?c)) then tons_of_plants1(?c) else 0.0]/total_neighbors_plants2),
     @left : if(swarm_location1_x == 1) then 0.0 else (sum_{?c : cell}[if ((swarm_location1_x - 1) == x_location(?c) ^
                                          swarm_location1_y == y_location(?c)) then tons_of_plants1(?c) else 0.0]/total_neighbors_plants2),
     @right :if(swarm_location1_x == 3) then 0.0 else (sum_{?c : cell}[if ((swarm_location1_x + 1) == x_location(?c) ^
                                          swarm_location1_y == y_location(?c)) then tons_of_plants1(?c) else 0.0]/total_neighbors_plants2));
swarm_location2_x= switch(swarm_move2){case @up: swarm_location1_x,case @down: swarm_location1_x,
                                              case @right: swarm_location1_x + 1,case @left: swarm_location1_x - 1};
swarm_location2_y= switch(swarm_move2){case @up: swarm_location1_y + 1,case @down: swarm_location1_y - 1,
                                              case @right: swarm_location1_y,case @left: swarm_location1_y};
swarm_location_x'=swarm_location2_x;
swarm_location_y'=swarm_location2_y;
tons_of_plants'(?c)=if (swarm_disabled) then tons_of_plants(?c) else (if (swarm_location2_y == y_location(?c) ^
                                              swarm_location2_x == x_location(?c)) then tons_of_plants1(?c)*0.5 else tons_of_plants1(?c));
swarm_disabled' = if (swarm_disabled) then swarm_disabled else exists_{?c : cell} ((disable_swarm_act(?c) == 1)^
                                              x_location(?c) == swarm_location2_x ^ y_location(?c) == swarm_location2_y);
};
 reward = [sum_{?c : cell}(tons_of_plants'(?c))];
}

non-fluents swarm_of_locusts2_nf {
     domain = swarm_of_locusts2;
     objects { cell : {cell1, cell2, cell3, cell4, cell5, cell6, cell7, cell8, cell9};};
non-fluents {
x_location(cell1)=1;x_location(cell2)=2;x_location(cell3)=3;x_location(cell4)=1;x_location(cell5)=2;x_location(cell6)=3;
x_location(cell7)=1;x_location(cell8)=2;x_location(cell9)=3;y_location(cell1)=3;y_location(cell2)=3;y_location(cell3)=3;
y_location(cell4)=2;y_location(cell5)=2;y_location(cell6)=2;y_location(cell7)=1;y_location(cell8)=1;y_location(cell9)=1;};
}

instance inst_swarm_of_locusts2 {
     domain = swarm_of_locusts2; non-fluents = swarm_of_locusts2_nf;
     init-state { swarm_location_x=2; swarm_location_y=2;};
     max-nondef-actions = 1; horizon  = 9; discount = 0.9;}
```

Acknowledgements

This work was supported by the Israel Ministry of Science and Technology Grant 3-15626, ISF Grant 1651/19

References