A Short Introduction to ABP

July 2010

This document illustrates use of our ABP library in an interesting two-dimensionsal hunting simulation involving two wolves hunting a rabbit. Our goal is to write a program where the wolves work together to trap the rabbit.

1 Wolves and Rabbits

Our game world's state is represented by a two-dimensional grid as in the class given below.

```
class Grid {
  Grid moveWolf1(Move m);
  Grid moveWolf2(Move m);
  Grid moveRabbit(Move m);

  boolean rabbitCaught();

  ...

  static Grid INITIAL;
}
```

This class maintains the coordinate positions of each animal on the grid, and implements game logic and rules. We omit some of the details that are not important for the following discussion. Changes to the state are made via three move methods `moveWolf1`, `moveWolf2`, and `moveRabbit`. Each of these methods returns a new `Grid` with the move applied. Additionally, a method `rabbitCaught` is given that checks to see if the rabbit has been captured in the current game grid. The static constant `INITIAL` corresponds to the instance of the world with the wolves at the top-left and the rabbit at the bottom right. This is used for the game's initial configuration.

Permissible moves for each animal are described via the `Move` enumeration.

```
enum Move {
  STAY,LEFT,RIGHT,UP,DOWN;

  static final Set<Move> SET =
    unmodifiableSet(EnumSet.allOf(Move.class));
}
```

The `Rabbit` class given below is an interface for any number of fixed strategies that the rabbit may take.

```
abstract class Rabbit {
  abstract Move pickMove(Grid g);

  static Rabbit random();
}
```

The static method `random` returns a rabbit that moves randomly.

A game proceeds as follows. First, the rabbit may observe the location of the wolves and then move one square in any direction. Next, the first wolf gets to observe the rabbit's move and move itself. After that the second wolf gets to observe both prior moves and then move itself. If at the end of a round either wolf has landed on the rabbit's square, the rabbit is captured. Otherwise, the hunt continues.

To make the problem more interesting we allow the x-coordinate of the world to wrap. Hence an animal at the left end of the grid can wrap around to the right end and vice versa. In this way, the rabbit could always escape if the wolves close from the same direction. Hence, the wolves must be smart enough to cooperate and close from opposite directions.

A programmer solving this problem with the above definitions might initially sketch out pseudocode such as that given below.

```
Rabbit rabbit = Rabbit.random();
Grid g0 = Grid.INITIAL;
while (true) {
  Move rabbitMove = rabbit.pickMove(g0);
  Grid g1 = g0.moveRabbit(rabbitMove);

  // ... pick move for wolf 1
  Move wolf1Move = ???
  Grid g2 = g1.moveWolf1(wolf1Move);

  // ... pick move for wolf 2
  Move wolf2Move = ???
  Grid g3 = g2.moveWolf2(wolf2Move);

  if (g3.rabbitCaught()) {
    ... raise our score a large amount
    break;
  }

  ... lower our score a small amount
  g0 = g3;
}
```

The grid `g0` represents the world state at the beginning of the loop each iteration. Again this includes the location of each animal. We move each animal as described before in the rules. The `Grids` `g1`, `g2`, and `g3` corresponds to the game state after each animal's move.

For now, we are uncertain about how to select the wolf moves so we indicate that with question marks. Near the end of the loop, we check to see if the rabbit has been captured. If not, we reassign `g0` and perform another round.

This pseudocode motivates some observations.

- There is a natural sense of constrained uncertainty when our wolves select their moves. They have to pick one of a small finite set of moves, but picking a good one might require a complex strategy and lots of code.
- We can consider the score as a reward or indicator of success or failure. It can be used as a metric telling us how successful our wolves are. Moreover, it suggests that it is easer to classify good and bad strategies, than to generate them.
- There is an inherent dependency between each wolf's strategy. They must work together. Moreover, the reward or score applies to both as they share a common goal.

1.1 Adaptation Concepts

The coupling between choice and reward suggests that some sort of abstraction could automatically select sequences of moves and then evaluate those moves by checking the score. Over time, this abstraction could identify better and better sequences of choices so as to optimize their average reward. Implementing these notions is the goal of our ABP library which we now describe.

We define an *adaptive variable* (adaptive for short) as one of these points of uncertainty in a program where we must make some decision amongst a small discrete set of choices. A value generated by one of these variables is called an adaptive value. We call the location of this uncertain selection a *choice point*.

An adapative can suggest a potential action at a choice point. But in order to do so, it requires some unique descriptor that identifies the state of the world. In our above example, there are two points of uncertainty, they are wolf move selection inicated with `???`. In our library we represent adaptive variables via the `Adaptive` class shown below.

```
public class Adaptive<C,A> {
  public A suggest(C context, Set<A> actions);
}
```

Adaptive values are parameterized by two type variables. The first (`C`) corresponds to the context or world state, and the second (`A`) corresponds to the type of permissible actions that the adaptive can take.

The context parameter `C` gives the library a clean and efficient description of what the world looks like at any given point. In our hunting example, this will initially be the `Grid` class.

The `suggest` method is our way of asking the adaptive for an appropriate value for some context. Moreover, we pass a set of permissible actions for the adaptive to choose from. We are asking the adaptive value, "If the state of the world is `context`, what is a good move from the set of `actions`?"

In our wolf hunt example, each of our wolves could be represented by an adaptive. The context type parameter would be the world state `Grid`, and the action type would simply be a `Move`. We illustrate this shortly.

The dependency between the moves we select for our wolves elicits another important observation: multiple adaptives might share a common goal. Under this view our adaptive wolves must share a common reward stream. The score in the game (the

reward) applies to both wolves, not just one. This sharing of rewards asks for a scoping mechanism that allows the grouping of multiple adaptive variables.

We define this common goal as an *adaptive process*. It represents a goal that all its adaptives share, it distributes rewards (or penalties) to its adaptives, and manages various history information that its adaptive variables learn from.

We show the basic interface in our ABP framework defining an adaptive process below.

```
public class AdaptiveProcess {
  static AdaptiveProcess init(File source);
  public <C,A> Adaptive<C,A>
           initAdaptive(Class<C> contextClass,
                        Class<A> actionClass);

  public void reward(double r);
  public void disableLearning();
}
```

A new adaptive process is created via the class's `init` method. The source file argument permits the `AdaptiveProcess` to automatically be persisted between runs. The first time the program is run, a new file is created to save all information about adaptives. When the program terminates, the process and all its adaptives are automatically saved. Successive program invocations will reload the process information from the given source file. This persistence permits the adaptive variables contained in the process to evolve more effective strategies over multiple program runs.

Individual adaptives are created with the `initAdaptive` method. The context and action type parameters are passed in as arguments, typically as class literals. This permits us to dynamically type check persisted data as it is loaded.

In our wolf example we would use the following code to initialize our process and the adaptives for each wolf.

```
public class Hunt {
  public static void main(String[] args){
    AdaptiveProcess hunt = AdaptiveProcess.init(new File("table"));

    Adaptive<Grid,Move>
      w1 = hunt.initAdaptive(Grid.class,Move.class),
      w2 = hunt.initAdaptive(Grid.class,Move.class);
```

We specify the aforementioned rewards of an adaptive process with the `reward` method. Positive values indicate positive feedback and tell the process that good choices were recently made, negative values indicate bad choices were recently made.

Upon receiving a reward, the adaptive process will consider the previous actions it has taken and adjust its view of the world accordingly. This permits later calls to `suggest` to generate better decisions. Details of the mechanics are described more formally in Section **??**. We discuss the final method of `AdaptiveProcess` `disableLearning` later.

Continuing with the previous block of code, the body of our game we sketched out earlier in pseudo code could be implemented as follows.

```
    Rabbit rabbit = Rabbit.random();
    Grid g0 = Grid.INITIAL;
    while (true) {
      System.out.print(g0);

      Move rabbitMove = rabbit.pickMove(g0);
      Grid g1 = g0.moveRabbit(rabbitMove);

      // ... pick move for wolf 1
      Move wolf1Move = w1.suggest(g1,Move.SET);
      Grid g2 = g1.moveWolf1(wolf1Move);

      // ... pick move for wolf 2
      Move wolf2Move = w2.suggest(g2,Move.SET);
      Grid g3 = g2.moveWolf2(wolf2Move);

      if (g3.rabbitCaught()) {
        // ... raise our score a large amount
        hunt.reward(CATCH_REWARD);
        break;
      }

      g0 = g3;
      // ... lower our score a small amount
      hunt.reward(MOVE_PENALTY);
    }
  }
}
```

The interesting pieces of this code are those near the comments where we filled in adaptive code and we discuss them here. First, the wolf strategies are as simple as calls to the adaptive variables' `suggest` methods. In each case, we pass in the current game state (the `Grid`) and the set of all moves `Move.SET`. Note that every move is legal; we simply translate moves into walls as `Move.STAY` for simplicity.

Second, where we referred to scores earlier, we place calls to the `reward` method of the adaptive process representing our hunting goal. After each unsuccessful round we assess a small penalty `MOVE_PENALTY`. Once the rabbit is captured we reward a large amount `CATCH_REWARD`. In our example we use -1 and 1000, respectively.

The print statements allow us to display the world before each step in a textual form. A The `.` represents an empty cell. A `1` indicates the position of the first wolf, `2` indicates the position of the second wolf. If the two wolves are in the same cell, we indicate it with a `3`. A `r` indicates the position of the rabbit.

If we run the game over multiple iterations, the average number of moves for the wolves to catch the rabbit drops fairly quickly. After a few thousand runs the average is between 6 and 7 moves.

With grid dimensions of 3×4 from the initial state, the worst-case optimal solution is 4 moves. That is, if the rabbit stays as far away from the wolves each round, it will take up to 4 moves to capture it. With a random rabbit, we should expect smaller averages.

The reason for this initial discrepancy is two-fold. First, a few thousand iterations is

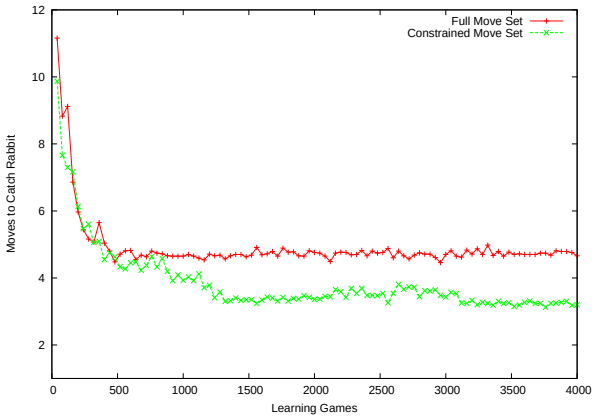

Figure 1: Average number of moves to catch the rabbit.

not a long time for the adaptive process to learn an optimal strategy; after a few hundred thousand games, the average drops much lower. Second, and more importantly, even once an optimal strategy is found, the adaptive process will continue to search for better ones.

Note, that an initially bad looking move, might lead to a better overall solution. Hence, it is necessary for the adaptive process to operate suboptimally as its adaptive variables try various move sequences.

Recall the `disableLearning` method of `AdaptiveProcess` whose description we deferred earlier. This method tells the adaptive process to suspend its search through suboptimal values and use only the best moves for every given world state. This is a means for the programmer to indicate to the adaptive process that it should stop searching and work deterministically with the knowledge it currently has. During practice we explore new strategies, but during the big game we use what we know works. If we transition to this optimal mode and run the wolf hunt program shown before, the average number of moves to catch the rabbit drops to around seven after just a few thousand rounds.

In Figure 1 the plot titled “Full Move Set” shows the average number of rounds necessary to catch the rabbit as a function of the the number of games learned when playing in this optimal mode. We discuss the second plot presently.

1.2 Optimizing the Adaptation Behavior

Recall in one extreme we consider algorithms that are fully deterministic, the programmer fully describes how to solve the problem and manually handles all cases. Such algorithms can be complicated even for easy tasks. In an opposite extreme we offload the entire strategy to some function that we learn via machine learning algorithms. But in doing so we forfeit control over various apsects of it. Furthermore, debugging such algorithms is difficult.

One of the goals of our ABP library is to offer a middle ground to this tension where the programmer can specify some of the analytical solution, but leave small pieces to

be learned. An example of this is shown by the `suggest` method of `Adaptive`. The second argument takes a set of permissible moves.

Let us suppose that our programmer implementing the wolf hunt game observed that any successful strategy would require the two wolves moving in opposite directions. In that case, we could easily specify this partial knowledge by passing in a subset of the permissible moves. We illustrate this by showing a slice of the earlier code inside the game loop.

```
// ... pick move for wolf 1
Set<Move> w1Moves = setOf(Move.LEFT,Move.DOWN,Move.STAY);
Move wolf1Move = w1.suggest(g1,w1Moves);
Grid g2 = g1.moveWolf1(wolf1Move);

// ... pick move for wolf 2
Set<Move> w2Moves = setOf(Move.RIGHT,Move.DOWN,Move.STAY);
Move wolf2Move = w2.suggest(g2,w2Moves);
Grid g3 = g2.moveWolf2(wolf2Move);
```

Including the above constraint gives the more efficient "Constrained Move Set" plot shown in Figure 1. (The `setOf` function just wraps a list into a `Set`.) This constrained learning permits a better solution, and one that can typically be learned faster since the adaptives must explore fewer alternatives.