Targeted Property-Based Testing

Andreas Löscher and Konstantinos Sagonas
Department of Information Technology, Uppsala University, Sweden

Setup

Assignment Files:

- Gist with `magic.erl`, `spells.erl`, and this assignment
- proper.tar.gz

Download the assignment files and unpack the linked version of PropEr. Build PropEr and set the `ERL_LIBS` environment variable to PropEr's path. Check that everything is installed correctly:

```
$ export ERL_LIBS=/path/to/proper/
$ erl
Erlang/OTP 20 [erts-9.0] [source] [64-bit] ...

Eshell V9.0  (abort with ^G)
1> l(proper).
{module,proper}
2> proper:module_info(compile).
[{options,[debug_info,
           {warn_format,1},
           warn_export_vars,warn_obsolete_guard,warn_unused_import,
           warn_missing_spec,warn_untyped_record,
           {i,"include"}]},
 {version,"7.1"},
 {source,"/path/to/proper/src/proper.erl"}]
3> c(magic).
{ok,magic}
4> c(spells).
{ok,spells}
```

Simulated Annealing

Simulated annealing (SA) is a local search meta-heuristic and the default search strategy used by PropEr in the extension for targeted property-based testing.

SA starts with a random initial input from the input space and sets it as currently accepted input. During each search step SA produces a random neighboring input and accepts it if the associated

fitness is higher than the one of the currently accepted input. It also accepts worse inputs with a probability that is dependent on the current temperature. The higher the temperature, the higher the probability that a worse input is accepted. The temperature typically decreases as the search goes on (see Wikipedia).

The neighboring input is generated by a **neighborhood function** that given a *base input* and a *temperature* produces a random input that is similar (in the neighborhood) to the base input. The quality of the neighborhood function is important for the efficiency of the search:

- If the neighborhood is very large the input space can be traversed fast, but the search might degrade to random testing. With a large neighborhood it can also take many steps to narrow down to a good input that is otherwise close in the input space of an already accepted input.
- If the neighborhood is very small, the search can usually narrow down fast input that is close/similar. However, it might require many steps to reach other parts of the input space or to escape local optima.

It is possible to reduce the size of the neighborhood function during the search using the temperature parameter. This can *sometimes* be useful. In many cases it is sufficient or better to keep the neighborhood size constant.

Magic

We want to design a role-play game (RPG) where our character has some attributes like strength and intelligence (see Wikipedia) that affect how well actions are performed. A strong character will typically be better at fighting and an intelligent character will be better at using magic.

A record stores the attributes for a player:

```
-record(attr, {strength     = 0 :: integer(),
               constitution = 0 :: integer(),
               defense      = 0 :: integer(),
               dexterity    = 0 :: integer(),
               intelligence = 0 :: integer(),
```

```
                charisma     = 0 :: integer(),
                wisdom       = 0 :: integer(),
                willpower    = 0 :: integer(),
                perception   = 0 :: integer(),
                luck         = 0 :: integer()}).
```

A newly created character gets some initial attributes:

```
-spec new_character() -> attr().
new_character() ->
  #attr{strength     = 5,
        constitution = 5,
        defense      = 5,
        dexterity    = 5,
        intelligence = 5,
        charisma     = 5,
        wisdom       = 5,
        willpower    = 5,
        perception   = 5,
        luck         = 5}.
```

Furthermore we want our RPG to have a spell-based system that allows us to customize these attributes. The player can cast a spell that will reduce some of these attributes and increase others. But spells are fragile things and can behave unexpectedly and wild when cast together with other spells. A seemingly useful spell can have dire consequences.

Our spell attribute API also contains the function `cast_spells(Attrs, Spell)` that calculates the resulting attributes after a list of spells is cast. If when casting a spell a player does not have enough points in attributes than need to be reduced, the spell has no effect on the attributes but might have othe hidden effects.

In `spells.erl` there is a list of spells for our RPG.

PropEr Spells

With the spells in place, how can we make sure that a player cannot exploit our spell system and get lets say get three times as good? We can specify a property to test for this type of exploit as follows:

```
prop_spells() ->
  ?FORALL(Spells, list_of_spells(),
          begin
            InitialAttr = spells:new_character(),
```

```
            BuffedAttr = spells:cast_spells(InitialAttr, Spells),
            SumAttr = spells:sum_attr(BuffedAttr),
            SumAttr < 3 * spells:sum_attr(InitialAttr)
          end).
```

We create a new character and then cast a random list of spells. Then we calculate the total number of attributes with `sum_attr(Attrs)` and check if the spells somehow managed to triple them compared to the initial attributes. The generator `list_of_spells()` looks like this:

```
list_of_spells() ->
  list(proper_types:noshrink(oneof(spells:spells()))).
```

Note that we don't want PropEr to shrink the values inside the spell records. We can prevent that by wrapping the generator with `proper_types:noshrink()`.

If we test this property now with PropEr, we will see that the property holds for all randomly generated inputs even if the number of tests is very big:

```
1> proper:quickcheck(magic:prop_spells(), 100000).
.... 100000 dots ...
OK: Passed 100000 test(s).
true
```

So all is well... only it actually ISN'T. There *does* exist at least one list of spells that can lead to such unlimited power.

Targeted Magic

We can change prop_spells() to make use of *a targeted* search strategy as follows:

```
prop_spells_targeted() ->
  ?FORALL_SA(Spells, ?TARGET(list_of_spells_sa()),
             begin
               InitialAttr = spells:new_character(),
               BuffedAttr = spells:cast_spells(InitialAttr, Spells),
               SumAttr = spells:sum_attr(BuffedAttr),
               ?MAXIMIZE(SumAttr),
               SumAttr < 3 * spells:sum_attr(InitialAttr)
             end).
```

We added the following elements:

- `FORALL_SA` specifies simulated annealing as search strategy
- `?TARGET` tells the strategy that it should generate input according to `list_of_spells_sa()`
- `?MAXIMIZE` specifies the search goal. In this case we want to maximize the attributes our character has after casting the spells.

Now we just need to tell simulated annealing how to generate the first element and which neighborhood function it should use:

```
list_of_spells_sa() ->
  #{first => list_of_spells(),
    next => fun list_of_spells_next/2}.
```

We can use the random generator `list_of_spells()` for the first element.

Task

Implement a neighborhood function in `list_of_spells_next()` for lists of spells.

The task is to implement the neighborhood function in `list_of_spells_next()` so that the property `prop_spells_targeted()` fails after a reasonable amount of time.

When writing the neighborhood function you can make full use of PropEr's language for defining custom generators (see PropEr API for defining generators)):

```
list_of_spells_next(PreviousSpells, Temperature) ->
  ?LET(SomeInteger, integer(), ...).
```

The first element of the neighborhood function is the base input. The temperature parameter is of type `float()` and decreases linearly from `1.0` to `0.0` with the tests.

After you have implemented the neighborhood function you can test the property with PropEr:

```
1> c(magic).
{ok, magic}
2> proper:quickcheck(magic:prop_spells_targeted(), 100000).
...
```

Make sure that with your neighborhood function finds a counterexample consistently and that the search does not get stuck for some runs. (You can check this by e.g. repeat the test for the property 100 times.)

Some remarks:

- Do not change the API in `spells.erl` or the property (`prop_spells_targeted()`)
- The whole input space must be reachable by consecutive calls of `list_of_spells_next()` . This means that lists should be able to grow and shrink in size.
- Use the temperature argument only if necessary.
- Even with targeted PBT the amount of tests needed can be in the few thousands but a counterexample should be found in under `20000` tests depending on your neighborhood function.
- The time to failure is more important than the amounts of tests needed to fail the property.
- If you neighborhood function does not perform good analyze what is going on:
  - Do the lists grow/reduce in size?
  - What is the value of `SumAttr` ? Maybe you just need a few more tests.
  - Can a neighbor delete spells at the end/start/middle of the spell list?
  - ...