Perkun

Dynamic Programming with Uncertainity

version 0.0.8

Pawel Biernacki <pawel.f.biernacki@gmail.com>

4 September 2015

Table of Contents

Abstract

Perkun is an experimental language for AI.

It can be downloaded from https://sourceforge.net/projects/perkun

It is an Open Source project under Gnu GPL version 3. See the file COPYING for more info about the licensing.

Beginning with the version 0.0.3 perkun comes as a library that can be used in external projects.

It contains a new algorithm – dynamic programming (Bellman) enhanced with the bayesian probability denoting the hidden variables values.

Building perkun

The following tools are necessary to build perkun:

- C++ compiler
- flex
- bison

See the file INSTALL for more info about the building. In short after unpacking the distribution package one should call:

```
./configure
make
make install
```

The last command may require superuser provileges.

Testing the build

After perkun is installed (the binary should be installed in /usr/local/bin/perkun) it is possible to launch the test cases. Type:

```
make check
```

Using perkun

The simplest way to use perkun is to run the perkun executable. For example if you have a file „specification.perkun” containing Perkun code you can run:

```
perkun specification.perkun
```

Using libperkun

If using autotools add the following line to your configure.ac:

```
PKG_CHECK_MODULES([PERKUN], [libperkun >= 0.0.5])
```

This will define PERKUN_LIBS and PERKUN_CFLAGS. In your C++ source code you should include the perkun header file:

```
#include <perkun.h>
```

Then you can inherit the class perkun::optimizer_with_all_data for example:

```
class myperkun: public perkun::optimizer_with_all_data
{
public:
	virtual void print_current_belief() const;
	virtual void get_input(std::map<perkun::variable*, perkun::value*> & m, bool & flag_eof);
	virtual void execute(const perkun::action * a);
};
```

When instantiated this class can be used to parse a perkun specification, for example:

```
myperkun x;
x.parse(”specification.perkun”);
```

It is likely you will want to use perkun for something else than command line execution, the simplest way to do it is to create two pipes and fork so that the child process (parsing the perkun specification) is using the pipes in the redefined virtual functions to communicate with the parent.

Optionally you can redefine the method:

```
virtual void optimizer::on_error_in_populate_belief_for_consequence(
	const belief & b1, const action & a, const visible_state & vs, belief & target) const;
```

By default it is throwing an exception if the visible state cannot be interpreted (its probability according to model was 0). By redefining it you can achieve a more gentle behaviour, for example making the target belief uniform.

Syntax

A Perkun program consists of the four obligatory sections followed by any number of instructions:

- values section
- variables section
- payoff section
- model section

Values section

The values section looks as follows:

```
values
{
	value v1, v2, v3;
}
```

The above example declares three values v1,v2 and v3. The variables declared in the following section will be allowed to have these values.

Variables section

```
variables
{
	input variable x:{v1, v2}, y:{v2, v3};
	output variable a:{v1,v2};
	hidden variable s:{v1,v2,v3};
}
```

The above example declares four variables. x and y are input variables, a is an output variable and s is a hidden variable. Each variable is followed by a list of values it may have. The list must not have duplicates. For example s may have one of three

values: v1, v2 or v3.

Perkun considers the so called hidden variables, i.e. the variables that are necessary to take into account when describing the world, but not directly observable to the agent.

Payoff section

```
payoff
{
	set({x=>v1,y=>v2},0.0);
	set({x=>v1,y=>v3},100.0);
	set({x=>v2,y=>v2},250.0);
	set({x=>v2,y=>v3},0.0);
}
```

The payoff section contains the so called payoff function defined as a float value for every single legal combination of the input variables values.

It is legal to omit some combinations, then the payoff function will have random values for them.

Model section

```
model
{
set({x=>v1, y=>v2, s=>v1},{a=>v1},{x=>v1, y=>v2, s=>v1}, 0.2);
set({x=>v1, y=>v2, s=>v1},{a=>v1},{x=>v1, y=>v2, s=>v2}, 0.0);
set({x=>v1, y=>v2, s=>v1},{a=>v1},{x=>v1, y=>v2, s=>v3}, 0.0);
set({x=>v1, y=>v2, s=>v1},{a=>v1},{x=>v1, y=>v3, s=>v1}, 0.3);
set({x=>v1, y=>v2, s=>v1},{a=>v1},{x=>v1, y=>v3, s=>v2}, 0.1);
set({x=>v1, y=>v2, s=>v1},{a=>v1},{x=>v1, y=>v3, s=>v3}, 0.0);
set({x=>v1, y=>v2, s=>v1},{a=>v1},{x=>v2, y=>v2, s=>v1}, 0.0);
set({x=>v1, y=>v2, s=>v1},{a=>v1},{x=>v2, y=>v2, s=>v2}, 0.1);
set({x=>v1, y=>v2, s=>v1},{a=>v1},{x=>v2, y=>v2, s=>v3}, 0.2);
set({x=>v1, y=>v2, s=>v1},{a=>v1},{x=>v2, y=>v3, s=>v1}, 0.0);
set({x=>v1, y=>v2, s=>v1},{a=>v1},{x=>v2, y=>v3, s=>v2}, 0.1);
set({x=>v1, y=>v2, s=>v1},{a=>v1},{x=>v2, y=>v3, s=>v3}, 0.0);
}
```

set

The set instructions in the model section have the following syntax:

```
set(INITIAL_STATE_QUERY, ACTION_QUERY, TERMINAL_STATE_QUERY,
TRANSITION_PROBABILITY);
```

The queries contain pairs variable=>value separated by commas in {}.

Initial state query

The initial state query must contain all input and hidden variables. It must not contain any output variables.

Action query

The action query must contain all output variables. It must not contain any other variables.

Terminal state query

The terminal state query must contain all input and hidden variables. It must not contain any output variables.

The sum of the probabilities over all terminal state queries must be 1.0 for each pair (initial_state_query, action_query).[1]

It is legal to omit some set instructions (or even all of them). The model will be random.

The transition probability must be a float value within 0.0 and 1.0.

illegal

Within the model section one can use the instruction „illegal”. It specifies that a certain action cannot be used in a given visible state.

```
illegal(VISIBLE_STATE_QUERY, ACTION_QUERY);
```

The visible state query should contain the input variables only. The action query should contain the output variables only.

impossible

Within the model section one can use the instruction „impossible”. It specifies that a certain state cannot be reached.

```
impossible(STATE_QUERY);
```

The state query should contain the input and hidden variables.

Instructions

cout << model << eol;

In order to dump a model to the standard output one should use the instruction:

```
cout << model << eol;
```

This is the easiest way to create a new model once the values and variables are defined.

Simply insert the empty model section and use the above instruction to dump the model.

1 Perkun displays a warning if the condition is not met.

Example:

```
# An example how to create a new model
values
{
	value v1, v2, v3;
}

variables
{
	input variable x:{v1, v2}, y:{v2, v3};
	output variable a:{v1,v2};
	hidden variable s:{v1,v2,v3};
}

payoff {}

model {}

cout << model << eol;

# end of example
```

The model printed to the standard output can be copied and pasted into the Perkun code (cout << model << eol; produces a valid collection of set instructions).

loop(depth);

The actual way to use Perkun is an interactive mode - after the model section type:

```
loop(3);
```

This instruction makes Perkun to enter the interactive mode. The parameter is the recursion depth used to select the optimal action. Don't make it too large or the computation may take very long!

In the interactive mode it first prompts what variables values are expected to be entered and waits for your input. In the above example it will expect values of the variables x and y (all input variables in the declaration order).

After the prompt type for example:
perkun> v1 v2

The Perkun will calculate its belief and the optimal action, then it will print both. The optimal action will be chosen so that according to the model it will maximize the payoff function. The maximization will concern the expected value of the payoff functions within the next n steps, with n being the parameter passed to the loop instruction.

What is the belief?

Belief is a probability distribution over all combinations of the hidden variable values. In the beginning Perkun chooses a uniform belief, but after each input the belief will be updated to reflect the interpretation of the input given the model.

cout << prolog generator << eol;

This instruction asks Perkun to generate a Prolog code that can be used to create a model.

cout << xml << eol;

This instruction asks Perkun to produce an XML document based on the specification. Using the xsltproc tool and the perkun.xslt stylesheet this XML document can be transformed into an HTML document.

cout << haskell generator << eol;

This instruction asks Perkun to generate a Haskell code that can be used to generate a model.

Model generators

The Perkun models tend to be big and Perkun is unfortunately too weak to express the model in a compact way. But there is somehing that can do it – Prolog and Haskell!

In order to create a perkun model one can also use the "cout << prolog generator << eol;" instruction. It produces Prolog code that can be modified manually to create a valid model.

Look at the example src/t4.perkun. It only contains values and variables (the payoff and model sections are empty). Then it contains the instruction cout << prolog generator << eol;

It can be used to produce the Prolog code src/t4.prolog.

Compare it with the code src/t5.prolog. It is enhanced by certain rules which express complex model relationships for this particular example.

If you have Prolog (we use SWI Prolog) you can run the t5.prolog to produce something like t5.perkun. It is a valid Perkun code with a zeroed payoff function and a model enhanced with the rules we entered manually into src/t5.prolog. Remember to set the payoff function before actually playing with it!

Look at the "real life example" - src/t6.perkun. It is src/t5.perkun with added a payoff function.

If you prefer Haskell use the Perkun instruction „cout << haskell generator << eol;” to create a Haskell model generator. Both in Prolog and Haskell you are supposed to add manually some code for your model. Look at the corresponding generator code to figure out what must be done[2].

2 Look for the phrase PLEASE INSERT YOUR CODE HERE.

Dynamic programming with uncertainity

The algorithm used in Perkun is Bellman's dynamic programming enhanced with the uncertainity. It can be compared with the minimax. These algorithms differ in the following way:

- in the minimax the players know exactly the state of the game
- in the minimax the moves consequences are deterministic
- in the minimax there are two players with the opposite payoff functions

In the Perkun's algorithm:

- the player does not know exactly the state of the game (becase of the hidden variables)
- the moves are stochastic (there is a distribution probability of the transitions)
- there is only one player

The algorithm is implemented in the class perkun::optimizer.