# Svarog - an AI oriented language

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

2021-05-07

# Chapter 1

# The algorithm

The algorithm introduced in Svarog resembles minimax, with the following substantial differences:

- unlike in minimax, in Svarog there is only one agent
- the environment in Svarog is stochastic, not deterministic
- the current state of the world is not completely known to the agent

## 1.1 Values

The values are constants. We will denote the set of values as $V$.
The state of the environment is described by the values of the variables (some of them being visible to the agent are called input variables, some of them, invisible, are called hidden variables). Also the agent's decisions are defined in terms of the variables' values (for the so-called output variables).

## 1.2 Variables

Both the agent's input information as well as its output information is described in terms of variables' values. In Svarog there are three fundamental kinds of variables:

- input variables
- hidden variables
- output variables

### 1.2.1 Input variables

The input variables belong to the environment's state variables. A mapping of input variables to the set of values is called a "visible state".
We will denote the set of input variables as $I$. A function: $f : I \to V$ is called a visible state. The set of visible states will be denoted as $VS$.

### 1.2.2 Hidden variables

The hidden variables belong to the environment's state variables, just like the input variables. The difference is that the hidden variables are not directly observable.
We will denote the set of hidden variables as $H$.

A mapping of both input variables and hidden variables to the set of values is called a "state". A state is any function: $f : I \cup H \to V$. The set of all states will be denoted as $S$. There is a natural mapping from the set of states $S$ to the set of visible states $VS$, i.e. for every state we can point to the visible state it belongs to. The visible state in question is for any state $f : I \cup H \to V$ such a function $g : I \to V$ that $\forall_{i \in I} g(i) = f(i)$.

### 1.2.3 Output variables

The output variables are controlled by the agent. A mapping of output variables to the set of values is called an "action". The set of all output variables will be denoted as $O$. An action is any function $f : O \to V$. The set of all actions will be denoted as $A$.

## 1.3 Payoff

The payoff is a certain function $p : VS \to \Re$. This function is constant and specified in a Svarog program. The objective of the algorithm is to maximize the expected value of the payoff function throughout the next $n$ steps.

## 1.4 Model

The model is a collection of probability distributions. Formally it is a given constant function $m : S \times A \to P$ with $P$ being a set of all probability distributions $f : S \to \Re$. It is therefore guaranteed $\forall_{f \in P} \sum_{s \in S} f(s) = 1$ and $\forall_{f \in P} \forall_{s \in S} f(s) \geq 0$. When referring to a model value, say $m(s_i, a)(s_t)$ we call the state $s_i$ the "initial state" and $s_t$ the "terminal state". The value $m(s_i, a)(s_t)$ is a probability that given the current state $s_i$ and performing the action $a$ the agent will encounter the terminal state $s_t$. We will write this value shortly $m(s_i, a, s_t)$.

## 1.5 Belief

For any visible state $vs \in VS$ we can define all the states belonging to it. Let's denote such states as $S_{vs}$. It holds: $\forall_{s \in S_{vs}} \forall_{i \in I} s(i) = vs(i)$, i.e. the state $s$ belongs to $S_{vs}$ if and only if it has the same value as $vs$ for every input variable $i \in I$. Let us introduce a class of new probability distributions, namely $r : S_{vs} \to \Re$ with $\forall_{s \in S_{vs}} r(s) \geq 0$ and $\sum_{s \in S_{vs}} r(s) = 1$. Such probability distributions will be called "beliefs". Let us denote the set of all beliefs for a visible state $vs$ with $B_{vs}$. At any moment the agent knows its visible state $vs$ as well as one of the beliefs from $B_{vs}$.

## 1.6 The algorithm internals

### 1.6.1 get optimal action

The function $goa$ ("get optimal action") returns for any belief from $B_{vs}$ and natural number called "depth" $n \in N$ the action $a \in A$ such that the function $gpevfc$ ("get payoff expected value for consequences") returns maximal value for this action. In other words *goa* returns *argmax* from *gpevfc*.

### 1.6.2 get payoff expected value for consequences

The function *gpevfc* ("get payoff expected value for consequences") returns for any belief $b$ from $B_{vs_i}$, any natural number called "depth" $n \in N$ and any action $a \in A$ a real value. The value is 0 if depth equals 0, otherwise it is

$\sum_{vs_t \in VS} cp(b, a, vs_t)(p(vs_t) + gpevfc(bfc(b, a, vs_t), n-1, goa(bfc(b, a, vs_t), n-1)))$

with $p$ - payoff, $bfc$ - belief for consequence (described later) and $cp$ - consequence probability (described later).

Therefore *gpevfc* is a recursive function constructing a game tree for subsequent moves of our agent. For each action it checks the whole spectrum of possible consequences, for each consequence ($vs_t$) it calculates its probability using *cp*. In a special case when $n = 1$ this expression equals

$\sum_{vs_t \in VS} cp(b, a, vs_t)p(vs_t)$

which we recognize as the expected value of the payoff function $E(p)$. It is interesting that the beliefs passed downwards the game tree are changing - the agents "thinks": given the current belief $b$, an action performed $a$ and assumed consequence $vs_t$ I imagine my interpretation (the future belief) to be $bfc(b, a, vs_t)$ and I assume I will from then on act optimally (using *goa*). Of course $b$ not only can but sometimes has to be different than $bfc(b, a, vs_t)$ which is a consequence of the fact that the two visible states $vs_i$ and $vs_t$ (called "initial visible state" and "terminal visible state") can be different.

### 1.6.3 belief for consequence

The function *bfc* ("belief for consequence") constructs a new belief, a new interpretation for a former belief $b$ from $B_{vs_i}$, an action $a \in A$ and an observation (visible state) $vs_t \in VS$.

$bfc(b, a, vs_t)(s_t) = \sum_{s_i} b(s_i)m(s_i, a, s_t)/T$

with model $m$ and normalization sum $T = \sum_{s_t \in S_{vs_t}} \sum_{s_i} b(s_i)m(s_i, a, s_t)$.

Note that sometimes the normalization can be impossible (when $T = 0$). Such situations are called "surprises" and normally the algorithm throws an error on them.

### 1.6.4 consequence probability

For any belief $b \in B_{vs_i}$, an action $a \in A$ and a visible state $vs_t \in VS$ this function constructs a real number expressing the probability that once we perform the action $a$ in the visible state $vs_i$ with the belief $b$ we will end up in the visible state $vs_t$. The value equals:

$\sum_{s_i \in S_{vs_i}} \sum_{s_t \in S_{vs_t}} b(s_i)m(s_i, a, s_t)$

## 1.7 Using the algorithm

When using the algorithm the agent maintains the current belief which is subsequently adjusted to the observations using the function $bfc$ (belief for consequence). The initial belief can be set to the prior value (for example the uniform belief). On every iteration the algorithm expects the values of the input variables, which determine the visible state.

```
do
{
std::map<variable*, value*> m;
bool eof = false;

get_input(m, eof);
if (eof)
        break;

visible_state * s = vs.get(m);

if (former_belief == NULL)
{
        set_apriori_belief(current_belief);
}
else
{
        current_belief = bfc(former_belief, former_action, s);
}
const action * a = get_optimal_action(current_belief, depth);
execute(a);
former_action = a;
former_belief = current_belief;
}
while (true);
```

The above pseudocode demonstrates the internals of the loop command in the actual implementation. Note that we use the function $bfc$ to build the current belief depending on the former belief, the former action and the current visible state.

# Chapter 2

# The Svarog language

Svarog is a strict-form, AI oriented programming language. It is NOT a general purpose programming language. Its main purpose is an optimization in a game with a single agent who performs certain actions subsequently, without any opponent. The environment in the game is stochastic and the agent does not perceive the complete information about the state of the world (some state variables remain hidden).

## 2.1 The program structure

A program written in Svarog consists of the following parts:

- the values section
- the variables section
- the knowledge section
- any number of commands

This is an example Svarog program:

```
values {
        value false, true;
}

variables {
        input variable alpha:{false,true};
        hidden variable beta:{false,true};
        output variable gamma:{false,true};
}

knowledge {
}
```

### 2.1.1 Keywords

action, actions, and, assert, belief, beliefs, case, class, complex, cout, eol, estimate, function, generator, haskell, hidden, if, illegal, impossible, initial, input, knowledge, loop, model, not, object, objects, on, or, output, payoff, perkun, precalculate, precalculated, probability, prolog, requires, result, return, rules, set, solve, state, states, terminal, test, too, value, values, variable, variables, visible, write, xml

### 2.1.2 Identifiers

A valid identifier is any word beginning with a letter (including underscore) and containing any number of letters or digits that is not a keyword.

### 2.1.3 Comments

A comment begins with # and ends with the end of line character.

### 2.1.4 String literals

A string literal is specified by the quotation marks "". There are no escape characters!

### 2.1.5 Float literals

A float literal is a sequence:

```
-?{DIGIT}+\.{DIGIT}+(e(\+|-){DIGIT}+)?
```

The scientific notation is supported, for example: 4.29497e+09.
Unary minus is supported (beginning from the version 0.0.2)!

The negative float literals are useful when defining the payoff.

### 2.1.6 Integer literals

An integer literal is any number of digits, optionally preceded with a minus character:

```
-?{DIGIT}+
```

## 2.2 Values section

The values section beginns with the keyword "values" followed by { and ends with }. It may contain any number of the statements:

```
value identifier1,identifier2,...;
```

The identifiers must be valid identifiers (not keywords) and must not occur more than once each of them.

Example of a values section:

```
values {
        value false, true;
        value none;
}
```

## 2.3 Variables section

The variables section begins with the keyword "variables" followed by { and ends with }. It may contain any number of variable declarations. There are three kinds of variable declarations:

- input variable declaration
- hidden variable declaration
- output variable declaration

### 2.3.1 Input variable declaration

The input variable declaration begins with "input variable" followed by any number of variable inner declarations separated by commas, terminated by semicolon. A variable inner declaration is a variable name (an identifier) followed by colon, followed by a list of variable values enclosed in {}. For example:

```
input variable a:{false,true},b:{false,true};
```

The above input variable declaration contains declarations of two input variables, "a" and "b", each of them having the values "false" or "true".

### 2.3.2 Hidden variable declaration

The hidden variable declaration begins with "hidden variable" followed by any number of variable inner declarations separated by commas, terminated by semicolon.

```
hidden variable c:{false,true},d:{false,true};
```

The above hidden variable declaration contains declarations of two hidden variables, "c" and "d".

### 2.3.3 Output variable declaration

The output variable declaration begins with "output variable" followed by any number of variable inner declarations separated by commas, terminated by semicolon.

```
output variable e:{false,true},f:{false,true};
```

## 2.4 Knowledge section

The knowledge section in a Svarog program contains any number of:

- functions
- "impossible" clauses

- "payoff" clauses
- "action" clauses
- precalculated clauses

### 2.4.1 Function

A function begins with the keyword "function" followed by an identifier (the function name), followed by a list of parameters enclosed in (), followed by the function body enclosed in curly brackets {}. The parameters in the list of parameters are separated with commas. Each parameter is an identifier preceded with the keyword "variable" or the keyword "value".

An example of a function:

```
function is_initially_in(variable where_is_x, value town)
{
        if (initial value where_is_x == town)
        {
                return true;
        }
        return false;
}
```

The function body consists of the commands "if" and commands "return". For each variable (either a parameter or directly accessed input or hidden variable) it is possible to access either its initial value or terminal value. The conditions in the "if" commands can use "and" and "or" operators, as well as "not" and round brackets ().

### 2.4.2 "Impossible" clause

Certain states are impossible. They are marked as such with the "impossible" clause. They will be omitted by the algorithm during the calculations.

An example of such clause is:

```
impossible "dorban and pregor are in the same town yet dorban cannot see pregor" {
        requires initial value where_is_dorban == initial value where_is_pregor;
        requires initial value can_dorban_see_pregor == false;
}
```

### 2.4.3 "Payoff" clause

Example:

```
payoff "payoff for winning a fight":100.0 {
        requires initial value has_dorban_won_a_fight == true;
}
```

The payoff clause specifies the payoff function value for a certain state. In the above example the value equals 100.0 "points" if the initial value of the variable has_dorban_won_a_fight equals true. The payoff clause body may contain any number of the "requires" clause.

#### "Requires" clause

The "requires" clause consists of the "requires" keyword followed by a logical expression (possibly using "and", "or" and "not") referring to the input variables' initial values. It is allowed to use here also the functions (defined elsewhere within the knowledge section).

In the payoff clause body one must not use the hidden variables. Payoff is defined in terms of the input variables only.

### 2.4.4 "Action" clause

An action clause consists of the keyword "action" followed by an action query, followed by colon, followed by the clause body enclosed in the curly brackets {}. Example:

```
action {optimal_action=>ask_pregor_to_follow_dorban}:{
# here comes the action clause body
}
```

The action clause body contains any number of the "case" clauses appropriate for this action.

#### "Case" clause

A "case" clause consists of the "case" keyword followed by a string literal describing the case (used only for debugging purposes) and a case clause body enclosed in the curly brackets {}. In the case clause body the following constructs are allowed (any number):

- requires clause
- illegal clause
- probability clause

#### "Requires" clause

The "requires" clause consists of the "requires" keyword followed by a logical expression (possibly using "and", "or" and "not") referring to the input variables' initial values. It is allowed to use here also the functions (defined elsewhere within the knowledge section).

#### "Illegal" clause

The "illegal" clause consists of the keyword "illegal" followed by a semicolon. This clause specifies that using the action is illegal in this situation. Example:

```
case "dorban does not see pregor" {
requires initial value can_dorban_see_pregor==false;
illegal;
}
```

#### "Probability" clause

A "probability" clause consists of the "probability" keyword followed by a string literal, followed by colon and float literal, followed by the probability clause body. The float literal passed here is the probability value (it must not be greater than 1.0).

```
probability "dorban with pregor succesfully attack the vampire" : 0.75{
        requires terminal value has_dorban_won_a_fight==true;
        requires initial value where_is_vampire == terminal value where_is_vampire
        requires initial value where_is_dorban == terminal value where_is_dorban;
        requires initial value where_is_pregor == terminal value where_is_pregor;
        requires terminal value can_dorban_see_vampire == true;
        requires terminal value can_dorban_see_pregor ==true;
        requires terminal value is_pregor_following_dorban==true;
}
```

The "requires" clauses in the "probability" clause body may refer both to the input and hidden variables, both initial and terminal values.

If multiple "probability" clauses are used within a single "case" clause their probabilities should sum up to 1.0.

### 2.4.5 precalculated clauses

These sections are intended to be created automatically by a precalculate command. They contain the rules precalculated for all valid visible states and discretized set of beliefs.

The syntax for the precalculated clause is:

```
precalculated(<depth>,<granularity>)
{
        <on visible state clauses>
}
```

#### "on visible state" clauses

These clauses denote the precalculated rules for a visible state specified by a query. It may contain either the information "too complex" (if the discretization of the beliefs set would be too complex) or a certain amount of "on belief" clauses. Each "on belief" clause contains an action to be executed for a given visible state and belief, precalculated for the given depth and granularity.

## 2.5 Commands

After the values section, variables section and knowledge section any number of commands follow.

### 2.5.1 Loop

The command "loop" asks svarog to enter the interactive mode. In this mode the user is first asked to type the input variables' values in the given order (separated by space characters) and then svarog prints out its belief and the chosen optimal action.

The command "loop" has the following syntax: The keyword "loop" followed by (, followed by integer literal, followed by ), followed by semicolon. The integer literal is the depth of the game tree passed to the "get_optimal_action" function.

Example:

```
loop(5);
```

### 2.5.2 Printing out the visible states

It is possible to print out the visible states to the standard output with the following command:

```
cout << visible states << eol;
```

### 2.5.3 Printing out the states

It is possible to print out the states to the standard output with the following command:

```
cout << states << eol;
```

### 2.5.4 Printing out the knowledge

It is possible to print out the knowledge to the standard output with the following command:

```
cout << knowledge << eol;
```

### 2.5.5 Printing out the result

In a single problem specifications (after the "solve" command) one can print out the result with the following command:

```
cout << result << eol;
```

### 2.5.6 Estimating the precalculations

For a given depth of the game tree and granularity (a small integer not smaller than 2, preferably 2) it is possible to estimate the amount of "on belief" rules in the precalculated knowledge. This is done by the command:

```
cout << estimate(<depth>,<granularity>) << eol;
```

The command does not produce a valid svarog specification (the "on belief" rules don't contain any actions).

### 2.5.7 Precalculations

There are two commands for the precalculations. The first one is precalculating knowledge for all visible states:

```
cout << precalculate(<depth>,<granularity>) << eol;
```

Note, however, that on machines with multiple processors it is not optimal to use this command. Instead it is better to divide the work among the processors using the help scripts written in Perl (svarog_generate scripts). They rely on a different precalculate command with the syntax:

```
cout << precalculate(<depth>,<granularity>,<query>) << eol;
```

The query (the third argument) denotes the visible state that should be precalculated.

### 2.5.8 test

There is a command "test" to verify whether the model specified by the knowledge is valid. The results will be printed out to the standard output. The syntax is:

```
test();
```

There is also a specialized version of the command with a query as a parameter. It can be used to test a particular visible state and state (determined by a single query).

### 2.5.9 input

The command "input" is used in the single problem specifications, in particular those sent to the svarog-daemons. The command specifies the visible state with a query.

```
input <query>;
```

### 2.5.10 belief

When a visible state has been specified (with the command input) we are free to specify a belief for this visible state with the command "belief".

```
belief {<belief case clauses>};
```

The belief case clauses have the form:

```
case <query>:<float>;
```

### 2.5.11 action

The command "action" is used along with the "input" and "belief" commands, in a single problem specifications. It specifies an action to be considered:

```
action <query>;
```

### 2.5.12 solve

The command "solve" is used to calculate a result in a single problem specification. Its parameter is the depth (the game tree depth, the same parameter as the one passed to the "loop" command).

```
solve(<float>);
```

# Chapter 3

# The tools

## 3.1 svarog

The svarog tool is an interpreter of the Svarog language. It is a program (written in C/C++ with Bison and Flex) that obtains as input a file name. This file should contain a valid Svarog specification.

### 3.1.1 Building the svarog tool

Download and unpack the svarog tarball from the website www.perkun.org. Enter the directory svarog. Execute the command:

```
./configure
```

#### Prerequisites

The following tools are necessary to build svarog from source:

- flex - a tool to generate scanners
- bison - a tool to generate parsers
- readline - a library

#### Compiling the svarog

Once configured, execute the command:

make

If no errors occur then execute:

```
sudo make install
```

### 3.1.2 Interactive mode

If a svarog specification contains the command loop then after execution it will enter the interactive mode.

## 3.2 svarog-daemon

It is possible to use other machines (or the local host) to improve the svarog performance. In order to do that a svarog-daemon can be used. It is a daemon started with the following syntax:

```
svarog-daemon --port <port> --password <password>
```

In order svarog to use the svarog daemons on some machines a special configuration file must be present. It is located in /home/$USER/.svarog/svarog.ini .

```
#
# This is a configuration file for the svarog program
#

#
# One can configure multiple servers which can be used to facilitate the computing
# The servers can be given tasks and they can be asked for responses.
#
servers
{
        server "localhost:12345" password "123456";
}
```

Warning: As for version 0.0.6 the svarog-daemons don't use the precalculated knowledge, which makes them less usable if you have done the precalculations.

## 3.3 svarog-dummy-client

svarog-dummy-client is a tool that sends a svarog specification to a svarog-daemon running on a specified machine.

It is invoked with the following syntax:

```
svarog-dummy-client <file> <server> <port> <password>
```

The specification (file sent to the server) must not contain the "loop" command, since it is interactive. An example of a specification that can be sent to a server is the file:

```
svarog/examples/example2_single_problem.svarog.
```

## 3.4 svarog-merge

svarog-merge is a tool that can be used to merge the precalculated knowledge from two files. It is invoked:

```
svarog-merge <file1> <file2>
```

For the precalculated knowledge of the same depth and granularity all the "on visual state" entries that are present in file2 but missing in file1 will be copied and the result will be printed out to the standard output.

## 3.5 The svarog generate scripts

There are four Perl scripts installed along with svarog. They are called:

- svarog_generate_control_shell.pl
- svarog_generate_merge_shell.pl
- svarog_generate_precalculate_shell.pl
- svarog_generate_precalculate_tasks.pl

In order to perform precalculations on a machine with multiple processors do the following:

- Create a new directory and enter it
- Copy a valid Svarog specification (without any commands) into it, for example as specification.svarog
- Copy specification.svarog to create_visible_states.svarog
- Append "cout << visible states << eol;" to create_visible_states.svarog
- Execute: svarog create_visible_states.svarog > visible_states.txt
- Calculate the amount of lines in visible_states.txt, for example with wc -l
- Execute: svarog_generate_precalculate_tasks.pl <depth> <granularity> specification.svarog visible _states.txt
- Execute: svarog_generate_precalculate_shell.pl <amount of visible states> > precalculate.sh
- Execute: bash precalculate.sh

The last step will start the precalculations. All processors should obtain a work to do. In order to control whether the operation terminated succesfully a control shell can be used:

- Execute: svarog_generate_control_shell.pl <amount of visible states> > control.sh
- Execute: bash control.sh

If the control shell reports no errors the calculations terminated succesfully. Then you will want to merge the Svarog specifications created in the directory KNOWLEDGE:

- Execute: svarog_generate_merge_shell.pl <amount of visible states> > merge.sh
- Execute: bash merge.sh

It should produce a file result.svarog containing all the precalculated knowledge for the given depth and granularity.

# Chapter 4

# The libsvarog library

The libsvarog library is built and installed automatically when the svarog tool is built and installed. In order to use the library in your C++ project add the following line to your configure.ac file:

```
PKG_CHECK_MODULES([SVAROG], [libsvarog >= 0.0.6])
```

Depending on your system you might need to set the environment variable PKG_CONFIG_PATH:

```
export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig
```

## 4.1 Using the libsvarog library in your C++ projects

In your Makefile.am located near the source code of your project you might need to specify:

```
AM_CXXFLAGS = @CXXFLAGS@ @SVAROG_CFLAGS@
```

You may also need to add the @SVAROG_LIBS@ to your x_LDADD variable provided the program you build is named x.

### 4.1.1 Including the svarog header file

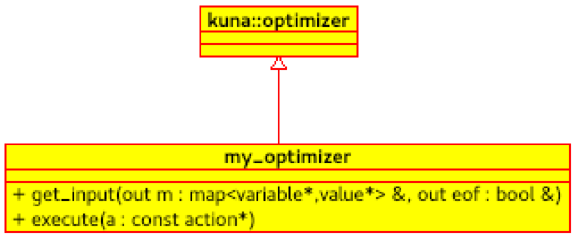
In you header file (for example x.h) you might want to include the svarog header file:

```
#include <svarog.h>
```

### 4.1.2 Inheriting the svarog::optimizer

You should define a new C++ class inherited from the svarog::optimizer class, for example:

```
class my_optimizer: public svarog::optimizer
{
public:
virtual void get_input(std::map<svarog::variable*, svarog::value*> & m, bool & eof);
virtual void execute(const svarog::action * a);
};
```


kuna::optimizer
my_optimizer
+ get_input(out m : map<variable*,value*> &, out eof : bool &)
+ execute(a : const action*)

As you can see above at least two functions should be redefined: get_input and execute.

### 4.1.3 Surprise handling

You might also want to redefine the virtual function:

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

The original implementation of this function throws an error. It happens when the belief (interpretation of the visible state vs) cannot be constructed, because the probability of the consequence vs was 0.0. You may want to replace this handling by a more gentle one, for example:

```
void my_optimizer::on_surprise(const svarog::belief & b1, const svarog::action & a
const svarog::visible_state & vs, svarog::belief & target)
{
        target.make_uniform();
}
```

You might also try to use the former belief b1 to construct a valid target belief.

## 4.2 Pipes and forks

It is recommended to embed Svarog in a multiprocess application with the pipes created for both directions communication and the Svarog interpreter (the class inherited from svarog::optimizer) running in a child process forked from the main process. The function my_optimizer::get_input should either set the eof flag to true or populate the map m with the appropriate values corresponding with the input variable values.

When instantiated, the new optimizer should call the function parse(const char * program) and if it succeeds (returns zero) then it should call the function run().

An example of such application is dorban.

# Contents