Triglav

triglav version 0.0.0
document version 0.0

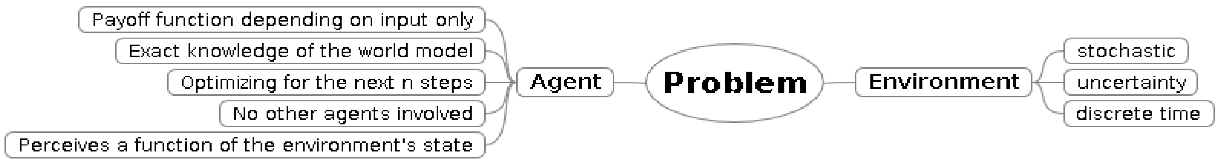
Pawel Biernacki

pawel.f.biernacki@gmail.com

Vantaa February 1, 2022

Chapter 1

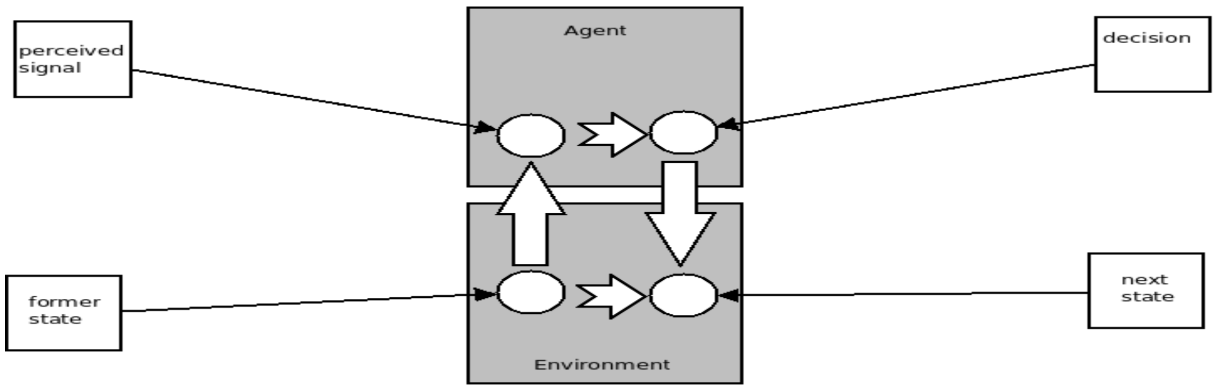
The optimization


Payoff function depending on input only
Exact knowledge of the world model
Optimizing for the next n steps
No other agents involved
Perceives a function of the environment's state
Agent
Problem
Environment
stochastic
uncertainty
discrete time

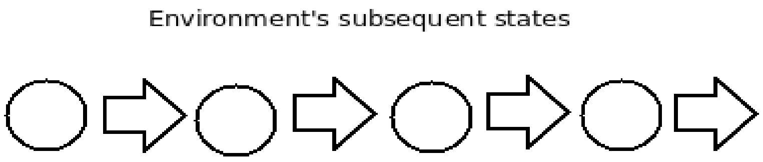
We are considering the following problem:

There is an stochastic environment with a certain state function in time and an agent which can perceive only a certain function of the environment state. The environment time is discret. In each step the agent can affect the environment and its future state depends stochastically on its former state and the agent's recent decision. Further we assume that the agent precisely knows the model of the world, i.e. the probability distribution denoting how the next state of the environment depends on the former state and the agent's action. Also we assume that the agent knows its payoff function which is constant and depends merely on the agent's input. Even though the agent knows the model of the world in terms of the exact environment states he never perceives the environment state itself, only a certain function of it.

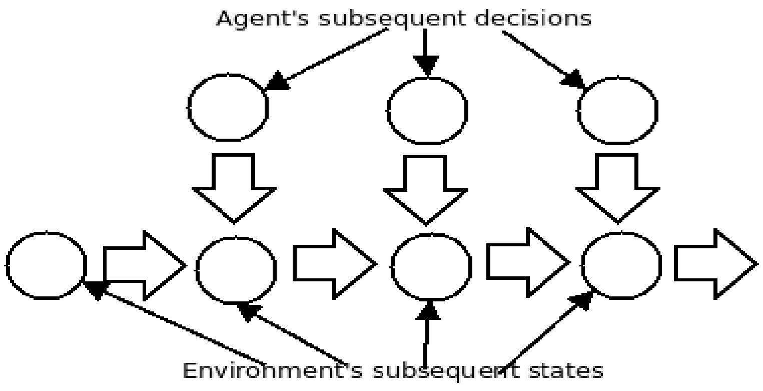
1.1 Environment and agent



This is the overall diagram demonstrating the dependencies between the agent and the environment in every single time step. We can see that the environment affects the agent providing him with the perceived signal through the mechanism of perception. The agent performs a decision making process which leads to a decision. The agent's decision and the former environment's state determine the probability of the next environment's state. The environment state will be called initial state, and the next state will be called terminal state.

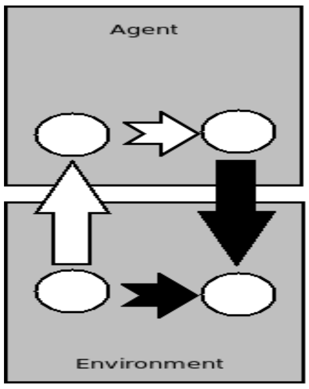


If we ignored the agent's existance then the environment would be a kind of automaton, with every single state in time depending stochastically on the former state. In addition we must consider the influence of the agent, therefore the dependencies from the environment's perspective look like this:



1.2 The agent

The agent as we stated before does not know the exact state of the environment, but knows exactly the model of the world indicated on the below diagram by the black arrows:



This knowledge is purely stochastic, i.e. the agent cannot predict precisely what will happen not only because he never knows the exact state of the environnment but also because of the stochastic nature of the environment.

It should be pointed out that the agent cannot use his model of the world directly.

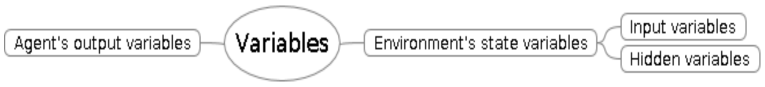
It may seem a little contradictory. What good is the world model for the agent if he does not perceive the actual state of the environment? The difficulty is that the agent could calculate the probability of the terminal state if he knew the initial state and assumed some decision. But he never knows the exact initial state. Wouldn't it be more logical to provide the agent with a model that tells him the dependency between the perceived states and the actions? The problem is as can be demonstrated for even simple games that the dependency between terminal state and initial state with the agent's decision cannot be expressed easily if the agent ignores what he cannot perceive.

The perception, however, gives the agent an idea about the possible state of the environment. It also plays an important role: the payoff function, i.e. the function we want to maximize in the next n steps depends only on the perceived state. The agent does not care what the real state of the environment is, he just likes some perceived states more than others, namely likes those that are likely to maximize the expected value of the payoff function.

Chapter 2

The problem

Here we will define the problem more formally.



We will introduce various variables describing the environment's states and the agent's actions. Please keep in mind that they are simply parameters mapping the environment states or agent's actions to a certain set V. These variables will not be modified during the problem solving in the algorithm that we propose. In some other algorithm the variables could be introduced dynamically.

2.1 The environment's states

Let us assume that the state of the environment is described by a number of variables. Without the limitation of the generality we can assume that some of the variables are perceived and some are not. Those that are perceived will be called the **input variables** (because they are the source of the input from the agent's perspective) and those that are not perceived will be called the **hidden variables**. Let us denote the set of the input variables with I and the set of the hidden variables with H. The set of the variable values will be denoted with V. We will further define any function of the form $f : I \cup H \to V$ to be an environment's state. The set of all such functions will be denoted S.

2.1.1 The perception

A function $vs : I \to V$ will be called a **visible state**. The set of visible states will be denoted VS. For any state $f : I \cup H \to V$ there is a natural mapping to a visible state. Formally there is a perception function $\psi : S \to VS$ such that for any state $f : I \cup H \to V$ it returns a function $g : I \to V$ satisfying the condition $\forall_{i \in I} g(i) = f(i)$.

2.2 The agent's actions

We also have certain actions that can be performed by the agent. These actions can be described in terms of variables, too, just like the environment states are described in terms of the hidden variables and the input variables. Let us denote the set of all the output variables with O. An **action** will be any function $a : O \to V$. The set of all such functions will be denoted as A.

2.3 The payoff function

The **payoff** is a certain function $p : VS \to \Re$. It is known to the agent and remains constant through the whole optimization process. This function determines what is "good" and what is "bad" for the agent. We can say that the payoff function creates a strong linear order on the set of visible states VS. A visible state vs_1 is considered "better" than vs_2 if $p(vs_1) > p(vs_2)$.

2.4 The model

The **model** is a function $m : S \times A \to P$ with P being a set of all probability distributions $f : S \to \Re$. The model is known to the agent and is constant. We do not consider the problem of improving the model throughout the time. For any state $s \in S$ and any action $a \in A$ the value $m(s, a)$ is a function denoting the probabilities of the agent's action a consequence when the "real" state of the environment is s. Because the value of m is a function will use the notation $m(a, s_i, s_t)$ instead of $m(a, s_i)(s_t)$ to denote the probability of jumping to the terminal state s_t when performing the action a in the initial state s_i.

2.5 The objective

The problem is to maximize the expected value of the payoff function throughout the next n steps. The number n is an integer known to the agent and is constant.

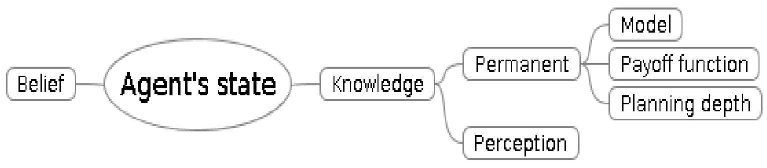
Chapter 3

The algorithms

In this chapter we will introduce the concept of a **belief**. We use the term as opposed to the term knowledge. The agent is free to maintain a state of his own, and this state can reflect his opinion about the possible state of the environment.

3.1 The belief

Here we propose a family of belief based algorithms which have the common feature of maintaining the agent's state so that he can estimate the probability of a certain actual state of the environment, or of a certain set of the actual states.

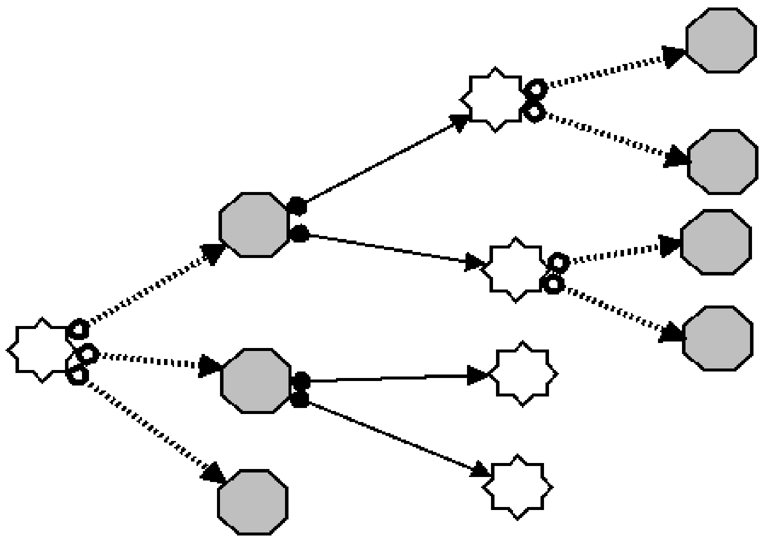


The belief based algorithms we propose here consist of two fundamental parts:

- planning
- application

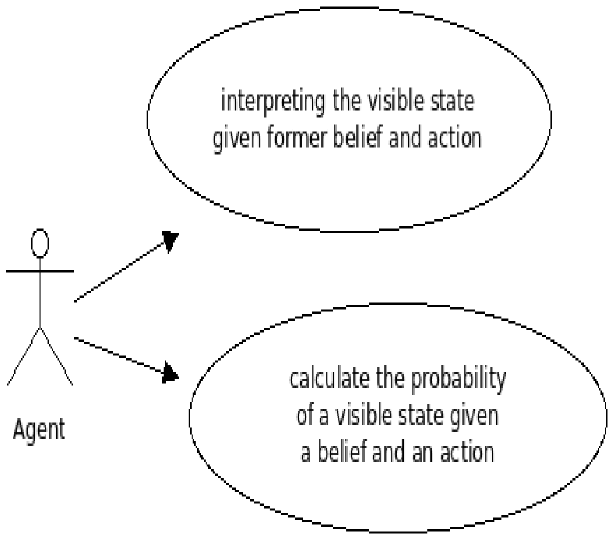
3.1.1 The planning

When planning we will construct a game tree of the following form:



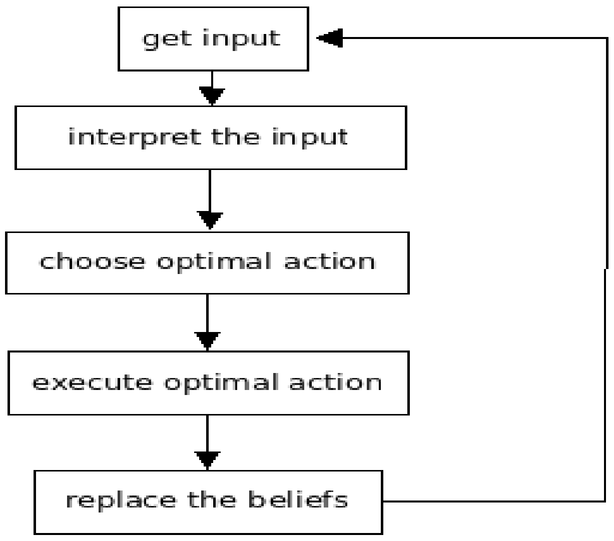
In these game trees there are two kinds of nodes and two kinds of edges, interchangeably. The white nodes are called **belief nodes**. In every belief node the agent has a certain belief with the root node corresponding to the actual belief he has about the current environment state. The grey nodes are **stochastic reaction nodes**. Whenever the agent considers a belief node he will have an action to perform. The edges connecting from belief nodes to the stochastic reaction nodes are agent's actions. Once an action is chosen the agent lands in a stochastic reaction node. Now it is time to predict various possible visible states (or sets of visible states) that may be perceived by the agent. For each possible visible state (or a set of visible states) the agent needs to interpret it, i.e. calculate the next belief. The edges connecting from stochastic reaction nodes to the belief nodes correspond with the reaction of the environment. Given the former belief, the action chosen and the visible state considered in the stochastic reaction node the agent must interpret this result, this outcome of his action by calculating somehow a new belief. This new belief represents his interpretation of what he would perceive in the future after performing the action a and observing the visible state vs. This belief, his interpretation of the hypothetical result must depend only on the former belief (beginning with the root belief), the action assumed and the visible state considered. In every belief node the agent knows what he would see (the visible state) and therefore is able to calculate the value of the payoff function. Then returning to the stochastic reaction node he will obtain a number of possible visible states with their interpretations (future beliefs) and the payoff function values. Now he has to calculate somehow the probability of each result (visible state). Combining these probabilities with the payoff function values allows calculating the expected value of the payoff function for every stochastic reaction node. The action chosen in the root should be this one which produces the maximal expected value.

There are two challenges in this approach. First we need to know how to interpret the hypothetical result after performing an action. Second we need to know how to calculate the probability of the visible state depending on what the agent knows and what he believes.


interpreting the visible state given former belief and action
Agent
calculate the probability of a visible state given a belief and an action

3.1.2 The application

Once we have a way to build a game tree and to choose the optimal action, i.e. an action that maximizes the expected value of the payoff function in the next n steps (n being the height of the game tree built in the previous step) we need to apply the algorithm in real life.



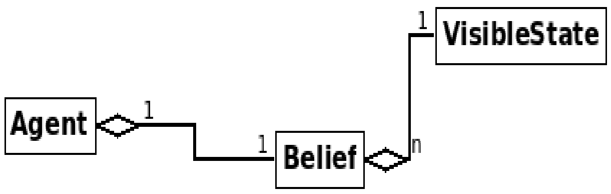
In the first step we get the input signal and identify the visible state determined by it. Then we interpret the input using exactly the same interpretation function we use in the planning. Then we build a game tree and choose the optimal action. Then we execute it and forget the former belief. Our current belief becomes now the new former belief and we go back to the first step.

3.2 Naive approach - the Perkun & Svarog beliefs

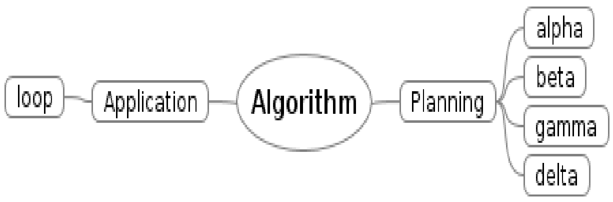
In Perkun and Svarog the belief is simply a probability distribution over the set of possible states. This is a naive approach, since the amount of states cannot be too large because of that. It works well, though, for small problems.

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 $b : S_{vs} \rightarrow \Re$ with $\forall_{s \in S_{vs}} b(s) \geq 0$ and $\sum_{s \in S_{vs}} b(s) = 1$. 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}, called "the current belief". For the sake of simplicity we will use the set B being the union of all sets B_{vs}.



Note that there is exactly one belief known to the agent (the current belief), and this belief implies a visible state. There can be many beliefs corresponding with the same visible state. Once we know the current belief we also know its visible state.



The algorithm, as mentioned before, consists of two parts. First we will learn how to build a game tree. In order to do that we will introduce four functions, α, β, γ and δ. Second, we will learn how the game tree is used in the algorithm's application.

3.2.1 get_optimal_action

We introduce a new function $\alpha : B \times N \rightarrow A$, i.e. for any belief $b \in B$ and any natural number $n \in N$ the value $\alpha(b, n)$ is an action. The semantics of this function is "get_optimal_action". Its second argument is the height of the game tree called the planning depth.
The value $\alpha(b, n) = argmax_{a \in A} \beta(b, n, a)$ with the function β defined later.

3.2.2 get_payoff_expected_value_for_consequences

Let us introduce a new function $\beta : B \times N \times A \rightarrow \Re$.
The function β ("get_payoff_expected_value_for_consequences") returns for any belief b from B, any natural number called "depth" $n \in N$ and any action $a \in A$ a real value. The value is 0 if $n = 0$, otherwise it is:

$\beta(b, n, a) = \sum_{vs_t \in VS} \gamma(b, a, vs_t)(p(vs_t) + \beta(\delta(b, a, vs_t), n - 1, \alpha(\delta(b, a, vs_t), n - 1)))$

with p - payoff, δ - belief for consequence (described later) and γ - consequence probability (described later).

Note that β is a recursive function. In each step of the planning we construct a new belief using the function δ, and the action assumed in the new belief node of the game tree is calculated using the function α which takes the arguments $\delta(b, a, vs_t)$ and $n - 1$.

3.2.3 consequence_probability

Let us introduce a new function $\gamma : B \times A \times VS \rightarrow \Re$. For any belief $b \in B$, an action $a \in A$ and a visible state $vs_t \in VS$ this function returns 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:

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

3.2.4 belief_for_consequence

The function $\delta : B \times A \times VS \rightarrow B$ ("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) $vs_t \in VS$.

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

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's implementation throws an error on them. It can be handled in a more smooth manner, though, when using Perkun or Svarog in your own applications.

3.2.5 Applying the algorithm

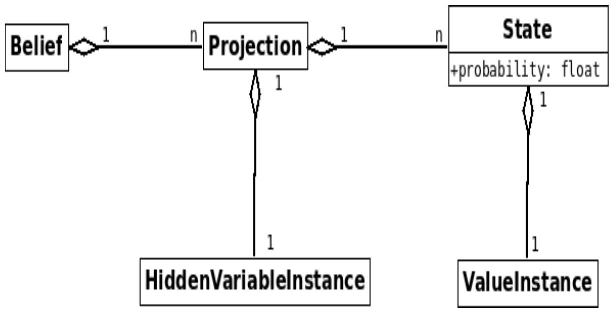
The algorithm application is performed in an infinite loop. It relies on the two functions defined above - α and δ. Please keep in mind that the agent maintains two beliefs: the former belief (b_0) and the current belief (b_1). It also maintains the former action a. The parameter n is a constant integer defined by the user, denoting the planning depth.

```
1. get new input $vs \in VS$
2. interpret the visible state $b_1 := \delta(b_0, a, vs)$
3. get optimal action $a := \alpha(b_1, n)$
4. update the belief $b_0 := b_1$
5. goto 1
```

As you can see we use the function δ both when planning (calculating the optimal action) and when applying the algorithm (to interpret the visible state). This step (step no.2) looks slightly different for the first run of the algorithm, namely we assign b_1 with a so called a-priori belief for the observed visible state vs.

3.3 Projections - the Triglav beliefs

In Triglav the belief structure is somewhat more complicated. Every belief has a number of projections, one for each hidden variable instance. In every projection there is a list of projection states corresponding with the variable values instances.



Unlike in Perkun and Svarog we do not really generate all the states as a cartesian product of the variable values sets. Instead we add a separate projection for each hidden variable instance. This allows operating on many more hidden variables than the naive approach used in Perkun and Svarog.

Chapter 4

The ideas

In this chapter we will describe some ideas we have put into the implementations, Perkun, Svarog and Triglav.

4.1 Impossible states

There are some methods to declare certain states as **imposible** in Perkun, Svarog and Triglav. This is an extension of the algorithm, since it does not take such impossible states into account.

4.1.1 Impossible states in Perkun

In Perkun there are four mandatory sections: values, variables, payoff and model. Within the model section one can use the "impossible" instruction, with a parameter being a query (enclosed in curly brackets) consisting of facts based on ALL input and hidden variables.

```
impossible({where_is_Dorban=>place_Wyzima, do_I_see_vampire=>false,
        where_is_vampire=>place_Wyzima});
```

The above code means that if both Dorban and vampire are in Wyzima, then Dorban cannot not see the vampire, i.e. he must see the vampire. This is game specific, in some other games it could be different. But in this game (Perkun Wars - a demo showing how to use Perkun in your own programs) it is like that.

4.1.2 Impossible states in Svarog

```
impossible "dorban can see pregor and ..." {
        requires initial value can_dorban_see_pregor == true;
        requires initial value can_dorban_see_pregor_is_alive
                == none;
}
```

The above ”impossible” clause taken from a knowledge section of a Svarog specification contains a keyword ”impossible” followed by a string literal (mandatory comment) and one or more ”requires” clauses in curly brackets. In this example we state that the initial value of the variable can_dorban_see_pregor_is_alive must not be ”none” when the initial value of can_dorban_see_pregor is true. This variable (can_dorban_see_pregor_is_alive) should be either true or false when Dorban can see Pregor, but not none (at least in this game).

4.2 Illegal actions

Some actions for certain visible states can be declared as ”illegal”. This narrows down the possible choice of actions given a visible state.

4.2.1 Illegal actions in Perkun

In Perkun, within a model section it is allowed to use the ”illegal” instruction, which takes two queries as parameters. The first one consists of facts based merely on input variables (not on hidden variables). It should contain all input variables. This query denotes a visible state. The second query denotes an action and it consists of facts based on output variables.

```
illegal({where_is_Dorban=>place_Wyzima, do_I_see_vampire=>false},
        {action=>goto_Wyzima});
```

The above example that if Dorban is in Wyzima and does not see the vampire, he may not go to Wyzima. This action is illegal.

4.2.2 Illegal actions in Svarog

The information that some actions are illegal are stored in Svarog within a mandatory ”knowledge” section. It may contains multiple ”action” clauses (one for each action). In the ”action” clause we may use a ”case” clause, which, along with the corresponding ”require” statements can be declared as illegal.

```
knowledge {
action {optimal_action=>ask_pregor_to_follow_dorban}:{
        case "dorban does not see pregor" {
                requires
                        initial value can_dorban_see_pregor==false;
                illegal;
        }
}
}
```

The above example is only a part of a knowledge section from a Svarog specification in Dorban (a Svarog demo). It says that Dorban (the character) cannot ask Pregor to accompany him if

Dorban cannot see Pregor. The case is provided with a text message, which can be empty, but it is advisable to use it as a comment - in this case the text is "dorban does not see pregor".

4.3 Precalculated knowledge

We have invented an optimization method implemented in Svarog (but not in Perkun), which is based on precalculating the optimal action with the given depth for various cases. A **case** is a tuple (vs, b), with vs being a visible state and b being a belief. (This is somewhat redundant, because the visible state is implied by the belief). It can be demonstrated that Perkun and Svarog behave differently in different cases, i.e. even for the same visible state they can make different decisions depending on what they believe in.

4.3.1 Precalculated knowledge in Svarog

There is a procedure how to precalculate the knowledge for Svarog. This procedure allows parallelizing the calculations which is particularly useful on machines with multiple processors. The calculations should be performed in a separate directory.

```
make PRECALC
cd PRECALC
```

Creating visible states

Copy a valid Svarog specification (without any commands, just with the mandatory sections) into a file specification.svarog. Then execute:

```
cp specification.svarog create_visible_states.svarog.
```

Append "cout << visible states << eol;" to create_visible_states.svarog.
Execute:

```
svarog create_visible_states.svarog > visible_states.txt
```

This should create a file with the visible states. Please calculate the amount of lines in the file visiblestates.txt:

```
wc -l visible_states.txt
```

In this example the amount of visible states was 360.

Creating the task scripts

```
svarog_generate_precalculate_tasks.pl 4 2 specification.svarog \
        visible_states.txt
```

This will create the ”task files” in the subdirectory TASKS, each task for a separate visible state. The first parameter (in the example above 4) is the planning depth. The second parameter (in the example above 2) is called granularity and it should be a small integer, preferably 2.

Create a precalculate shell script

```
svarog_generate_precalculate_shell.pl 360 > precalculate.sh
```

This will create a shell script ”precalculate.sh” calling the tasks (parelelly). Remember to replace 360 in the above code with the actual amount of visible states.

Execute the precalculations

Execute the script created previously with the command:

```
bash precalculate.sh
```

This step can take a long time to perform. The script returns immediately, but the tasks will be performed in the background.

Control the status of the precalculations

In order to create a control bash script execute:

```
svarog_generate_control_shell.pl 360 > control.sh
```

Then to test the status of the precalculations execute:

```
bash control.sh
```

If no errors were reported then the precalculations have been performed succesfully.

Merging the results

Create a merging bash script with the command:

```
svarog_generate_merge_shell.pl 360 > merge.sh
```

Performing the merging with the command:

```
bash merge.sh
```

Merging is not paralellized, but it should be relatively fast. It should produce a file result.svarog containing all the precalculated knowledge for the given planning depth and granularity.

4.3.2 Precalculated knowledge in Triglav

In Triglav there are usually many input and hidden variables instances. The space of their possible values combinations is therefore huge and we cannot precalculate the optimal action for each case. Instead we use the technique that was already used in *some* cases in Svarog, namely we create a "base" (consisting of the orthogonal vectors) and make the precalculations only for these vectors. Still, the amount of data produced will be huge.

Generating cases

In order to generate all the cases use the command:

```
generate_cases(1, 144);
```

It should follow the type definitions as well as variable definitions and "expand" command.

The first argument should be a small positive integer, for example 1. It denotes how many unusual values may be included in the generated cases.

The command produces files with the names cases_i.txt, with i being a number from 1 to 144 (the amount of case files in the above example).

WARNING: Executing this command may require a long time. In case of the example1.triglav it was about 40 days on my machine.

Creating a databank

In this step we create an XML file called databank. It is describing the precalculated knowledge files.

The command creating such a databank is:

```
save_databank("readme.xml", 144);
```

See the example4.triglav file in the examples folder. It contains the type definitions as well as the variables definitions as well as an "expand" command. Only after expanding the types and variables one can save a databank.

The first argument is the file name, the second is the amount of the cases files generated in the former step.

Validating cases

This step means checking which of the visible states for the generated cases are valid. Usually only items from a certain subset of VS are valid according to the rules in the Triglav specification (see the examples in the "examples" directory).

You should first find out how many logical processors per machine/how many machines you will have.

You can use the Perl script triglav_generate_validation_lists.pl (installed with triglav) to create files with the name "validation_list_i.txt" with i being an integer from 1 to the amount of the logical processors (if you use a single machine).

The below code is taken from the example5.triglav file. It opens a databank (an XML file "readme.xml" created in the previous step) and executes the command **validate_cases** with the argument 12. It is suitable for a single machine with 12 processors. In this code the parent process forks creating 12 children processes, each taking the corresponding validation list and performs validation of the visible states.

As a result triglav creates the files "valid_visible_states_i.txt".

```
open_databank("readme.xml");
validate_cases(12);
```

Precalculating

In this step we perform the precalculations, i.e. we calculate according to the triglav algorithm what is the optimal action to perform in the given case belonging to a valid visible state.

You should first open an existing databank and then use the command precalculate_cases. It takes the amount of processes as the first argument. Triglav forks that many child processes and precalculates the optimal actions for the planning depth passed as a second argument.

```
open_databank("readme.xml");
precalculate_cases(12, 3);
```

In this example the amount of processes is 12 and the planning depth equals 3.

Note that if you pass for example the planning depth 3 and you have the precalculated knowledge files for the planning depth 2 (or 1) in place then Triglav will use this precalculated knowledge rather than calculating the optimal actions. This means it might be reasonable first to make the precalculations for the planning depth 1, then 2 and so on.

4.4 Cartesian expressions

We introduce a new form of expressions. It is not only about syntax, but also the underlying mechanism will be explained.

```
type person={Gotrek, Gerrudir, Gwaigilion};
type place={Krakow, Warszawa, Wroclaw};
type information={(X:person)_is_going_to_(Y:place)};

expand(2);
```

The expression in the above example has the form:

```
(X:person)_is_going_to_(Y:place)
```

The formal grammar of such expressions is:

```
cartesian_expression : list_of_generic_name_items
list_of_generic_name_items : generic_name_item
                             list_of_generic_name_items
                             | generic_name_item
generic_name_item : T_STRING_LITERAL
                             | '(' placeholder_name ':' type_name ')'
placeholder_name : T_STRING_LITERAL
type_name : T_STRING_LITERAL
```

They will be called **cartesian expressions** since they represent a set of names, which can be expanded (through the "expand" command) for all the tuples from the cartesian product of the types involved.

For example in the type information from the above code the cartesian product of types person and place is used, which results in the following tuples:

```
(Gotrek,Krakow)
(Gotrek,Warszawa)
(Gotrek,Wroclaw)
(Gerrudir,Krakow)
(Gerrudir,Warszawa)
(Gerrudir,Wroclaw)
(Gwaigilion,Krakow)
(Gwaigilion,Warszawa)
(Gwaigilion,Wroclaw)
```

They "generate" (or match) the following enumerations (when expanded with the parameter 2 or greater):

```
Gotrek_is_going_to_Krakow
Gotrek_is_going_to_Warszawa
Gotrek_is_going_to_Wroclaw
Gerrudir_is_going_to_Krakow
Gerrudir_is_going_to_Warszawa
Gerrudir_is_going_to_Wroclaw
Gwaigilion_is_going_to_Krakow
Gwaigilion_is_going_to_Warszawa
Gwaigilion_is_going_to_Wroclaw
```

Note that it is allowed to use the same placeholder many times, for example:

```
(X:person)_can_see_(X:person)
```

In such case the placeholders should have the same type (in the example above they both belong to the type "person").

When applying a cartesian expression we denote a whole family of enumerations or variables. For example in Triglav we can define the variables in the following way:

```
type person={Gotrek, Gerrudir, Gwaigilion};
type place={Krakow, Warszawa, Wroclaw};
type information={(X:person)_is_going_to_(Y:place)};
type boolean={false,true};

input variable (X:person)_tells_me_(Y:information):boolean;
hidden variable (X:person)_tells_(Y:person)_(Z:information):boolean;

expand(2);
```

4.4.1 Declaring variables with "usually"

When declaring a variable we can add after the type name a keyword "usually" and a list of the usual values in curly brackets. This is important for the performance.

```
hidden variable (X:person)_tells_(Y:person)_(Z:information):boolean
        usually {false};
```

When performing the precalculations this information will be used to minimize the amount of the possible states. Instead of generating all the states we will perform the precalculations only for certain subset of it.

4.5 Recursive enumerations

There is a new idea concerning the knowledge representation implemented in Triglav. Both Perkun and Svarog are based on enumerative types. In Triglav we introduce a concept of **recursive enumerations**. For example:

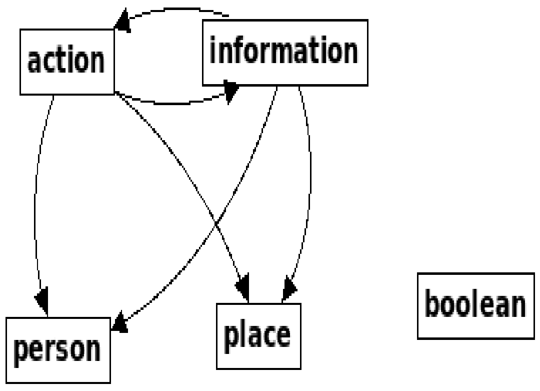
```
type person={Gotrek, Gerrudir, Gwaigilion};
type place={Krakow, Warszawa, Wroclaw};
type information={(X:person)_is_going_to_(Y:place),
 (X:person)_has_told_(Y:person)_(I:information)};
type boolean={false,true};

expand(3);
```

We can see from the above example that one of the information type values depends on the information itself. This is a recursion. We cannot expand it infinitevily, but we can expand it to a certain extent. This is done in runtime with the command "expand". It has one parameter, an integer literal, which denotes how deeply we expand the recursion.

It is also allowed that two (or more) type values refer mutually to each other, like in the below example:

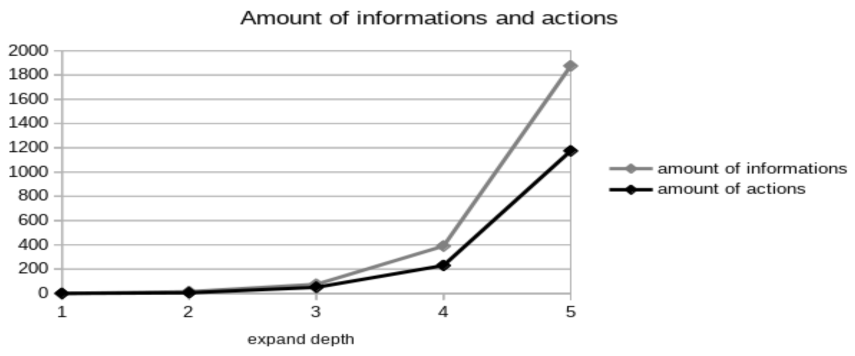
```
type person={Gotrek, Gerrudir, Gwaigilion};
type place={Krakow, Warszawa, Wroclaw,Poznan,Gdansk};
type action={go_to_(X:place),
        tell_(X:person)_(I:information)};
type information={(X:person)_thinks_that_(I:information),
        (X:person)_is_in_(P:place),
        (X:person)_has_done_(A:action)};
type boolean={false,true};

expand(3);
```

We can see that one of the actions depends on the information type, and one of the informations depends on the action type.



Let us see how the amount of "informations" and "actions" depends on the expand depth in this case:

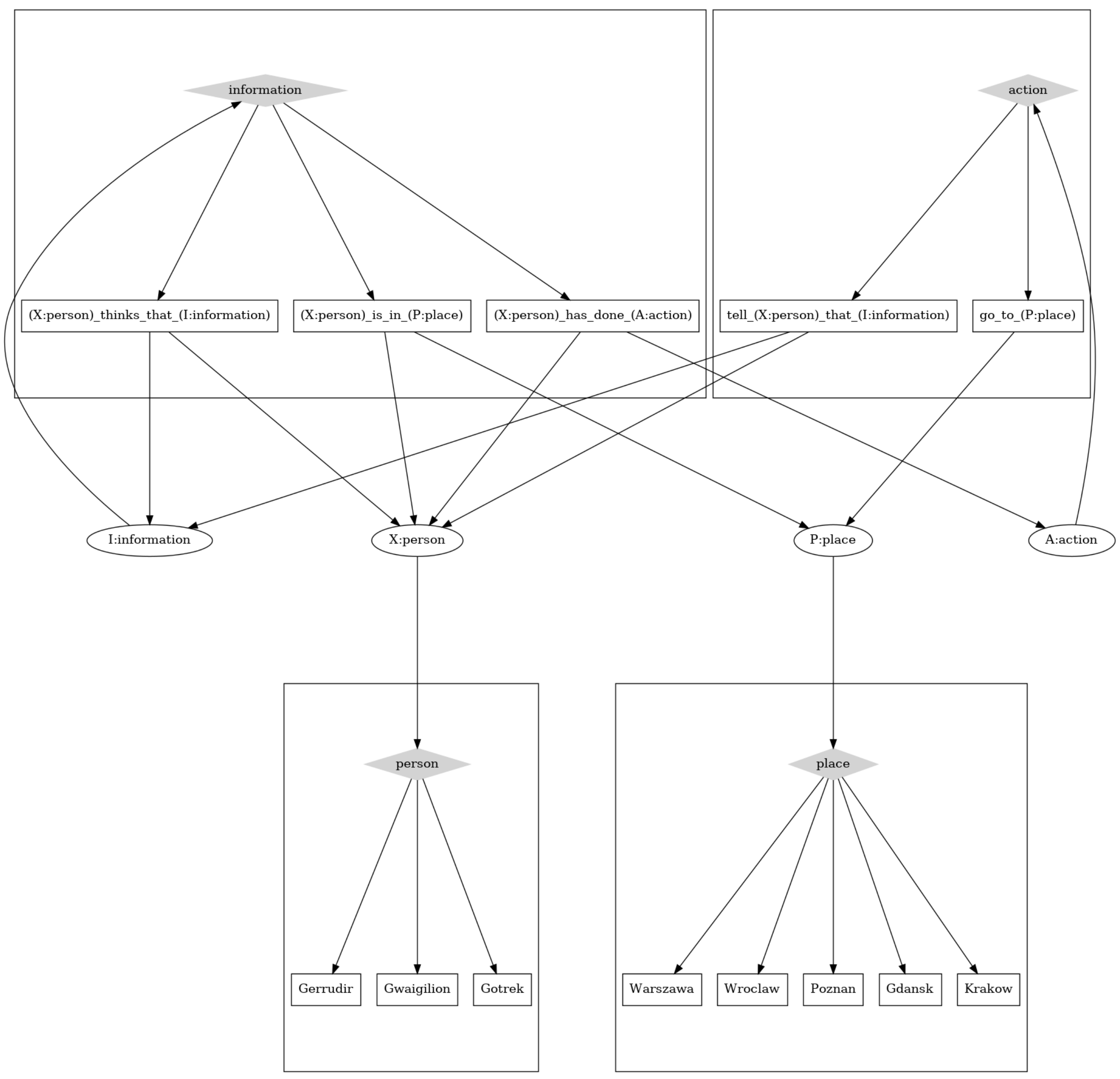
expand depth	amount of informations	amount of actions
1	0	0
2	15	5
3	75	50
4	390	230
5	1875	1175



As an example let us consider the below information value (produced with expand(4)):

```
Gwaigilion_has_done_tell_Gerrudir_Gwaigilion_is_in_Krakow
```

It means "Gwaigilion has told Gerrudir that Gwaigilion is in Krakow".



The above diagram shows the dependencies with types, values and placeholders. The placeholder nodes are marked with ellipsis, the type nodes are diamonds filled with the light grey color. The types "person" and "place" are simple enumerations, while the types "information" and "action" involve using the cartesian expressions and depend mutually on each other.

This is only an example, the dependencies can be much more complex. The actual complexity is controlled by the depth of the expanding operation, i.e. the integer parameter passed to the expand command.

4.6 Knowledge representation

Given the cartesian expressions and the recursive enumerations we introduce the following way of knowledge representation for simple facts.

```
initial value <CEVAR>==<CEVAL>
terminal value <CEVAR>==<CEVAL>
```

With <CEVAR> being a cartesian expression representing a variable (hidden, input or output) and <CEVAL> being a cartesian expression representing a value (one of the enumerations belonging to the type of the variable). We can apply these logical expressions to impose the conditions on the families of the variables.

It is also possible to set the conditions on the different initial values and terminal values of two variables:

```
initial value <CEVAR1>==initial value<CEVAR2>
terminal value <CEVAR1>==terminal value<CEVAR2>
initial value <CEVAR1>==terminal value<CEVAR2>
terminal value <CEVAR1>==initial value<CEVAR2>
```