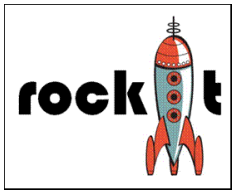


A fast Markov Logic solver

Documentation

written by
Jan Noessner
jan@informatik.uni-mannheim.de

Lehrstuhl für Künstliche Intelligenz
Prof. Dr. Heiner Stuckenschmidt
University of Mannheim

March 2014

Contents

Chapter 1

Installation Instructions

1.1 Instant use of rockIt on our Web-Platform (no installation required)

1.1.1 Start immediately

You can use rockIt without any installation requirements from our platform
`http://executor.informatik.uni-mannheim.de/systems/rockIt/`.
Read the user manual (Chapter 2) if you need help how to configure the input files.

1.1.2 Access our Web Plattform From Your own Software Using Rest Interfaces

Our platform also allows the use of rest interfaces. These allow you to start experiments out of your own code. Further information how to use these rest-interfaces can be found at
`http://executor.informatik.uni-mannheim.de/documentation/index`.

1.2 Use rockIt on Your Own Machine (Easy Installation)

This is the easy step by step installation instruction for Linux and Windows taking approximately 15 minutes. The steps are described in detail to minimize frustration.

1.2.1 Download Source-Files

Download the zip archive at `http://code.google.com/p/rockIt/downloads/list` containing the Jar-Files. It also contains the `rockIt.properties` file which is needed for MySQL configuration.

1.2.2 Install MySQL

For running rockIt you need the open-source MySQL database.

Linux For Debian/Ubuntu:

```
apt-get install mysql-server
```

For Arch Linux:

```
pacman -S mysql
```

In the following command prompt type in the root passwort "mannheim1234". (database root password). If you want you can also install a new user and tell the rockIt system its username and password in the rockIt.properties file:

```
sql_username = root
sql_password = mannheim1234
sql_url = jdbc:mysql://127.0.0.1/
```

Windows Install MySQL (http://www.mysql.com/downloads/), after running the MySQL installation, start the Configuration Wizard. Activate "Include Bin Directory in Windows Path" during configuration.

Adapt the file rockIt.properties to your local configuration. You can find the file in the rockIt directory in the zip-archive. The default settings are:

```
sql_username = root
sql_password = mannheim1234
sql_url = jdbc:mysql://127.0.0.1/
```

1.2.3 Install Gurobi

For solving integer linear programs, we use the ILP solver Gurobi. For academic purposes Gurobi offers a free license.

Register

Register at http://www.gurobi.com/. You need to give a valid mail address from a university. Login to your account.

Download

Download the newest Gurobi version for your specific platform (click on the checkbox with the ”readme” file) and install it.

Settings

Linux Set the following environment variables (add for example in the following lines to the hidden .bashrc in your home directory):

```
export GUROBI_HOME="/home/ubuntu/gurobi461/linux32"
export PATH="${PATH}:${GUROBI_HOME}/bin"
export LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:${GUROBI_HOME}/lib"
```

Windows Make sure that the environment variable grb_home is set correctly and that the bin directory of your Gurobi installation is included in the ”path“ environment variable.

License

Order a Free Academic License in your Gurobi account (account from step 2.1, click on `Licenses` and `Free Academic` and follow the instructions). Execute (via command line) the command you got from the Free Academic Licence. It says something like: `grbgetkey xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx`.

In case you store the license key to a non default location, follow this instruction: You have saved your license key to a non-default location. You will need to set environment variable GRB_LICENSE_FILE to value `your/directory/gurobi.lic` before you can use this license key.

Provide Jar File

Copy the gurobi.jar file from your GUROBI/lib directory to the rockIt/lib directory. Rename the file to ”gurobi-rockIt.jar”.

1.2.4 Test the rockIt Installation

In order to test the correct rockIt Installation, run the following command (one line):

```
java -jar rockIt-0.1.jar
-input data/TUFFYATVLDB2011/smoke/prog.mln
-data data/TUFFYATVLDB2011/smoke/evidence.db
-output out.db
```

1.3 Maven Repository

Before you can import the maven project, you have to install MySQL (Section 1.2.2) and Gurobi (Section 1.2.3).

You can deploy rockIt directly into your projects using our repository. You just have to add the following line to your `pom.xml` to access our repository.

```
<repositories>
   <repository>
        <id>lski</id>
        <url>https://breda.informatik.uni-mannheim.de/nexus/content/groups/
   </repository>
</repositories>
```

Then you have to add the rockIt dependency to your dependencies in your `pom.xml`. Please check the website `https://breda.informatik.uni-mannheim.de/nexus/index.html#nexus-search;quick~rockIt` which is the newest version of rockIt and replace the `XXX` below with the version number (e.g. `0.3`).

```
<dependency>
  <groupId>com.googlecode.rockIt</groupId>
  <artifactId>rockIt</artifactId>
  <version>XXX</version>
</dependency>
```

The last step you have to do is to manually install the gurobi artifact. Therefore, you need the path to the jar file, which is located in the `lib` folder of your gurobi directory of Section 1.2.3). For this task you have to execute the following command from your command line (written in one single line). Please make sure that you include your maven installation in the build path.

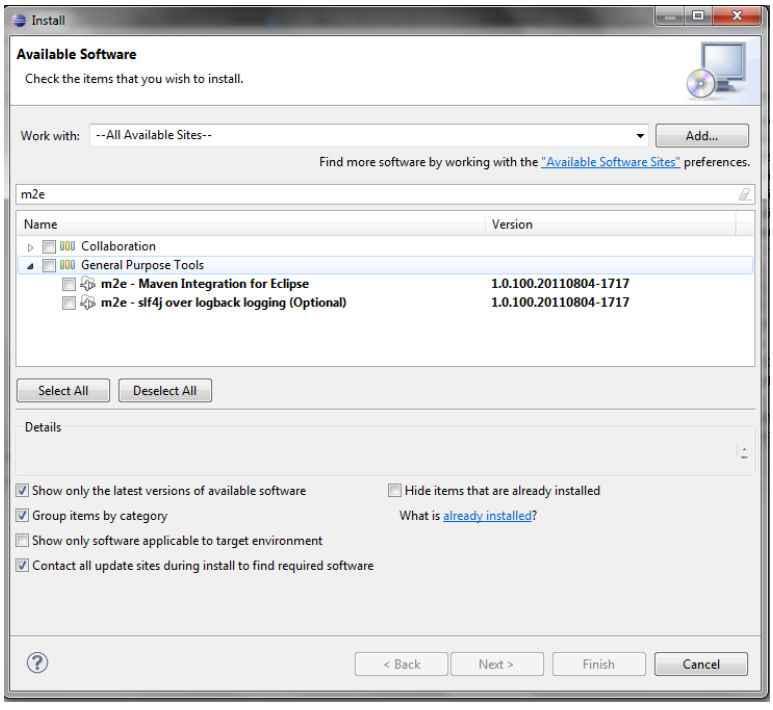
```
mvn install:install-file  -Dfile=path/to/your/rockIt/lib/rockIt.jar
                          -DgroupId=com.gurobi
                          -DartifactId=gurobi
                          -Dversion=rockIt
                          -Dpackaging=jar
```

1.4 Get the Source Code (for Developers)

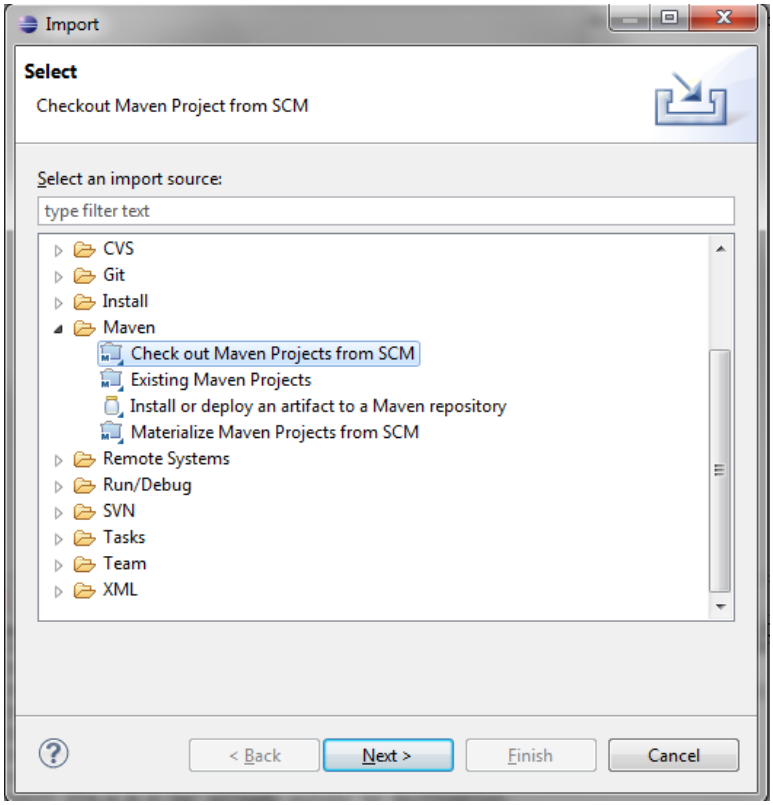
The project is a Maven project. Never heard of Maven? Do not be afraid to use it. We will provide a step by step installation instructions to Maven beginners for eclipse.

Before you can import the maven project, you have to install MySQL (Section 1.2.2) and Gurobi (Section 1.2.3). This installation instruction works for Eclipse Indigo or newer versions. Older versions of Eclipse do for instance not support the market place.

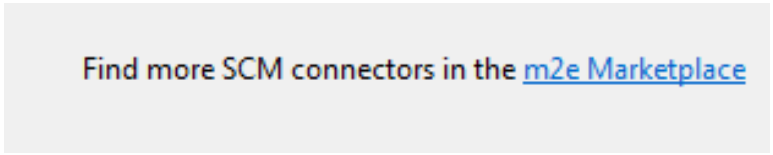
1. Install m2e in Eclipse (Help - Install Software). Alternatively, search for "maven".



2. Checkout Maven Project (Rightclick, Import...).

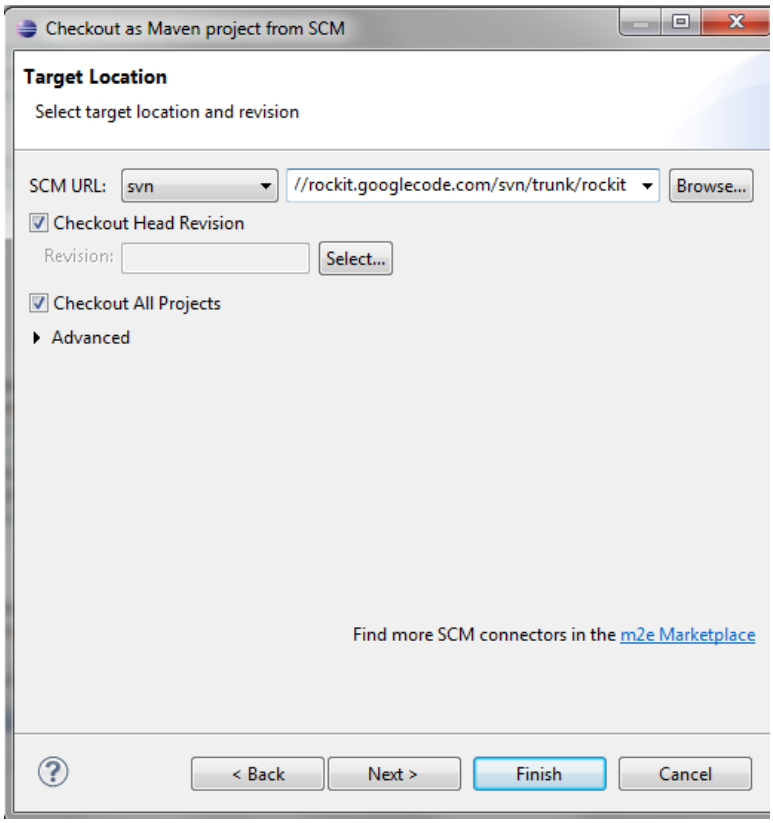


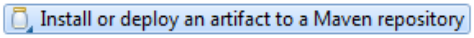
3. Click Next and then click on *find more Connectors*.



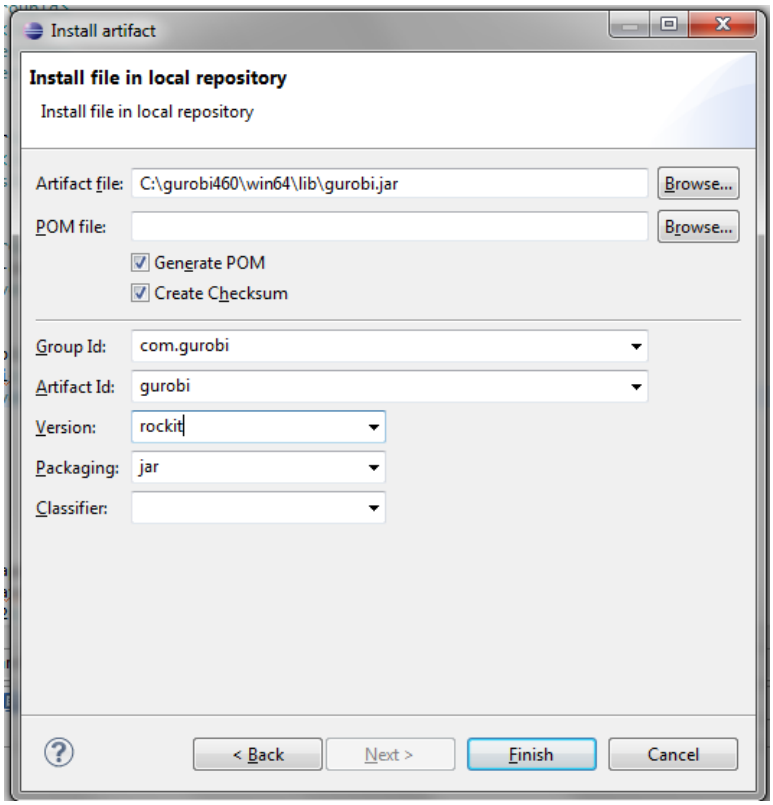
4. Choose m2e-subclipse.

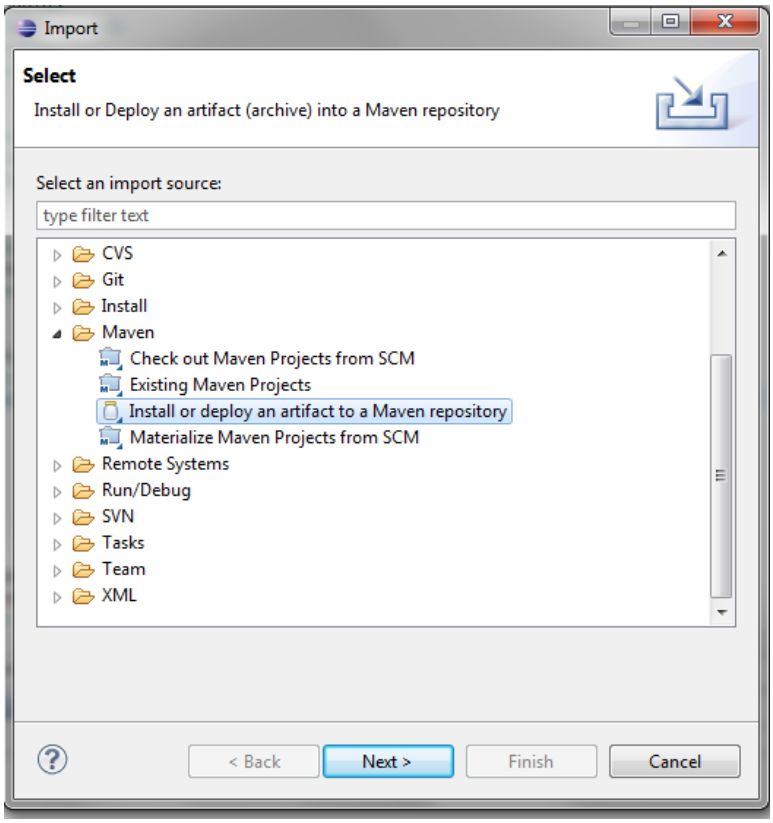
5. Checkout Project (at rockIt.googlecode.com/svn/trunk/rockIt).



6. Install Gurobi Artefact (Rightclick - Import... 

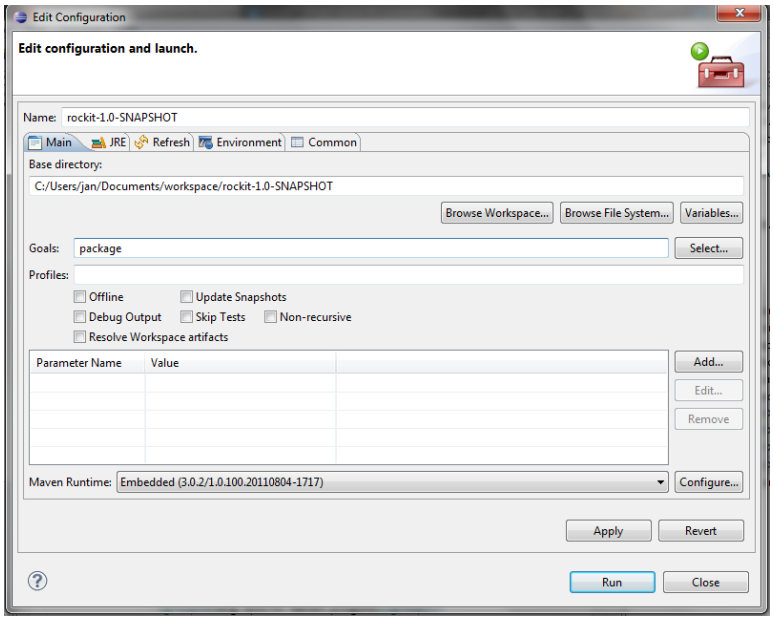
). It is important to change the version to `rockIt`! Even if the field may be gray it is possible to edit the version.


Install artifact
Install file in local repository
Install file in local repository
Artifact file:
C:\gurobi460\win64\lib\gurobi.jar
Browse...
POM file:
Browse...
Generate POM
Create Checksum
Group Id:
com.gurobi
Artifact Id:
gurobi
Version:
rockit
Packaging:
jar
Classifier:
< Back
Next >
Finish
Cancel



7. Update Dependencies (Rightclick on project - Maven - Update Dependencies)

8. Add package generation as run configuration



9. Now you can start to work with the source code. The Main class is located in `com.googlecode.rockIt.app.Main`. In order to test if rockIt is configured correctly, you can execute the class

   `com.googlecode.rockIt.test.MiniTestApplication`.

10. If you run the main class you might get an error which has to do something with the Gurobi licence (probably in Linux systems only). In that case you have to specify the environment variable `GRB_LICENCE_FILE`. To that end, go to your run configuration, click on the `Environment` tab and create a new variable (`New ...`) named `GRB_LICENCE_FILE` which points to your Gurobi license file.

Chapter 2

User Manual

2.1 Inference

rockIt supports the two main inference types in Markov logic - MAP inference and marginal inference. The input for both is probabilistic knowledge (soft formulas, refer to Section 2.3.1) and deterministic knowledge (hard formulas, existential formulas, and cardinality formulas, refer to Section 2.3.2, 2.3.3, and Section 2.3.4).

2.1.1 Maximum A-Posteriori (MAP) Inference

In *maximum a-posteriori (MAP) queries* (also called *most probable explanation*) the task is to find the most probable world given evidence. Let $\mathbf{E} = \mathbf{e}$ be the given evidence. Then, we aim to find an assignment of all non-evidence variables $\mathbf{X} = \mathbf{x}$ so that

$$\underset{\mathbf{x}}{\operatorname{argmax}} \; P(\mathbf{X} = \mathbf{x} \mid \mathbf{E} = \mathbf{e}).$$

We will often call evidence variables *observed variables* and non-evidence variables *hidden variables*. The assignment $\mathbf{x}$ which lead to the maximal P is called *maximum a-posteriori (MAP) state*. If more than one assignment leads to the same result, we can pick a random one. Finding such an assignment is not a trivial problem, since the assignment where a single potential function picks its most likely value is often not the optimal assignment for the global Markov network.

For solving MAP queries we apply Cutting Plane Inference (Riedel 2008, Improving the Accuracy and Efficiency of MAP Inference for Markov Logic) and Cutting Plane Aggregation (Noessner 2013, rockIt: Exploiting Parallelism and Symmetry for MAP Inference in Statistical Relational Models).

The MAP inference is the standard inference method of rockIt. Here you do not have to define anything. Most of the following examples in this documentation

target MAP inference. However, every example also works for marginal inference.

2.1.2 Marginal inference

The *(conditional) probability query* computes the posterior probability distribution

$$P(\mathbf{X}|\mathbf{E} = \mathbf{e})$$

over the values $\mathbf{x}$ of (a subset of) all variables $\mathbf{X}$ given the evidence values $\mathbf{e}$ of $\mathbf{E}$. This inference type is also referred to as *marginal inference*.

We implemented a simple GIBBS sampler which starts with the MAP state as consistent world. In each iteration we take one literal ℓ. Then, we swap ℓ and check if any hard constraint, any cardinality constraint, or any existential constraint is violated. If not, we swap ℓ with the probability:

$$p = \frac{\exp \phi^-(\ell)}{\exp \phi^-(\ell) \ + \ \exp \phi^+(\ell)}$$

where

- $\phi^+(\ell)$ is the sum of the weight of the true clauses containing ℓ if ℓ is *not* swapped and
- $\phi^-(\ell)$ is the sum of the weight of the true clauses containing ℓ if ℓ is swapped.

Efficient data structures and intelligent storing of the results makes our GIBBS sampling very fast and memory efficient. In particular, we use data structures in which we receive the clauses containing ℓ in constant time and, thus, are able to $\phi^+(\ell)$ and $\phi^-(\ell)$ very efficiently. Furthermore, we developed an efficient storage system of the results. We only document in which sampling round a specific literal ℓ changes. With this information we can compute the probability (of conjunctions) of literals in a straight forward way. At the same time we save memory because we do store the state of every literal after each sampling round.

In the near future, we will extend the GIBBS sampler to leverage symmetries as described in (Niepert 2013, Symmetry-Aware Marginal Density Estimation).

For activating marginal inference, you have to set the `-marginal` flag, when you execute rockIt (`java -jar rockIt.jar -marginal ...`). If you want, you can set the number of iterations (e.g `java -jar rockIt.jar -marginal -iterations 1000000 ...`). If not set, the number of iteration is 100 * (number of variables).

2.2 Get Started

Our Syntax is in most cases equivalent with the Alchemy[1] and the TUFFY[2] Syntax. If you are familiar with these Syntaxes, it should be easy for you to create models for rockIt. Please note, that the following examples are explained on the basis of the MAP inference query. However, every syntactic elements that are presented can also be applied with marginal inference.

2.2.1 Difference to Alchemy and Tuffy

For experienced MLN users which have used Alchemy or Tuffy in the past, we point out the differences in the syntax between our system RockIt and Alchemy. As inexperienced user we forward to Section 2.2.2.

RockIt's syntax is almost identical to that of existing Markov logic systems such as Alchemy and Tuffy. The main differences are:

- A predicate definition preceded by * is considered to have the closed world assumption; i.e., all ground atoms of this predicate not listed in the evidence are false. (Example: *friends(Human,Human)). We will call this special predicate an *observed* predicate since all ground atoms are known. All other predicates are called *hidden* or *query* predicates. The definition is possible in Tuffy, but not in Alchemy. It is always a good idea to define predicates as observed predicates, if no new knowledge should be inferred, since they are processed more effectively than hidden predicates.

- In RockIt, we do not need to define query predicates explicitly, because every predicate not preceded by * is a query predicate.

- Variable names can also start with a capital letter. Constants must be enclosed with double quotation marks (Example: "Constant"). Example of two equivalent formulations of a formula in:

  Tuffy and Alchemy:

  ```
  0.4 !Friends(x,Bob) v Smokes(y)
  ```

  RockIt:

  ```
  0.4 !Friends(x,"Bob") v Smokes(Y)
  ```

  The large `Y` can also be written as a small `y`.

[1] http://alchemy.cs.washington.edu/
[2] http://research.cs.wisc.edu/hazy/tuffy/

- In RockIt, it is not possible to use implications (=>) and conjunctions (^). The user has to transform her formula to CNF (`http://en.wikipedia.org/wiki/Conjunctive_normal_form`) such that they only contain disjunctions (v). Example:

  Tuffy and Alchemy:

  ```
  0.4 Smokes(x) => Cancer(x)
  ```

  RockIt:

  ```
  0.4 !Smokes(x) v Cancer(x)
  ```

- The syntax of existential formulas differs from the syntax of Tuffy and Alchemy. We refer the reader to Section 2.3.3 for details.

2.2.2 Simple Example

We explain the basics of our syntax with the well-known standard smoke example from [2]. In this example, we want to find out which persons probably have chancer. We assume that smoking causes cancer up to a certain degree, and that friends of each other are more likely to smoke. In Markov logic syntax the following equations are given:

$$\langle Smokes(x) \Rightarrow Cancer(x), 0.5\rangle \Leftrightarrow$$

$$\langle \neg Smokes(x) \vee Cancer(x), 0.5\rangle\,. \tag{2.1}$$

Formula 2.1 means that smoking will cause cancer with weight 0.5. We transformed the formula into conjunctive normal form.

$$\langle Friends(x,y) \Rightarrow (Smokes(x) \Leftrightarrow Smokes(y)), 0.8\rangle \Leftrightarrow$$

$$\langle \neg Friends(x,y) \vee \neg Smokes(x) \vee Smokes(y), 0.4\rangle \tag{2.2}$$

$$\langle \neg Friends(x,y) \vee Smokes(x) \vee \neg Smokes(y), 0.4\rangle \tag{2.3}$$

Formulas 2.2 and 2.3 express that if two persons are friends and one friend smokes, it is likely (with weight 0.8) that the other friend smokes as well.

Furthermore, we know the following facts (we will call them *ground values* from now on):

$$Friends(Anna, Bob)$$
$$Friends(Anna, Edward)$$
$$Friends(Anna, Frank)$$
$$Friends(Edward, Frank)$$
$$Friends(Gary, Helen)$$
$$Smokes(Anna)$$
$$Smokes(Edward)$$

The translation to the rockIt syntax is straight forward. As in other Markov logic systems like Alchemy, the definition of the formulas is separated from the groundings of these formulas. We first define the model with the formulas in the file `prog.mln`.

Listing 2.1: Model Definition of Smoke Example

```
// Predicates
// observed predicates
*Friends(person, person)
// hidden predicate with some ground values given.
Smokes(person)
// hidden predicates
Cancer(person)

// Formulas

// Smokes(x) => Cancer(x)
0.5  !Smokes(x) v Cancer(x)

// Friends(x, y) => (Smokes(x) <=> Smokes(y))
0.4  !Friends(x,y) v !Smokes(x) v Smokes(y)
0.4  !Friends(x,y) v Smokes(x) v !Smokes(y)
```

Furthermore, we define the ground values in the file `evidence.db`.

Listing 2.2: Ground Value Definition of Smoke Example

```
Friends(Anna, Bob)
Friends(Anna, Edward)
Friends(Anna, Frank)
Friends(Edward, Frank)
Friends(Gary, Helen)
```

```
Smokes(Anna)
Smokes(Edward)
```

When we now compute the maximum a-posteriori (MAP) query with rockIt we get the result:

```
Smokes("Anna")
Smokes("Edward")
Smokes("Frank")
Smokes("Bob")
Cancer("Anna")
Cancer("Edward")
Cancer("Frank")
Cancer("Bob")
```

In this simple example, everyone who smokes is also supposed to get cancer, because the weight of Formula 2.2 and Formula 2.3 is positive.

In the following subsections, the components of the model (file `prog.mln`) and the ground values (`evidence.db`) will be explained. The model files consists of the predicate definitions and of one or many formulas.

2.2.3 Predicate Definitions

Predicates are the components which are used in the formulas to describe the domain of interest. In our small example, the predicate `Smokes(person)` indicates that a person x smokes. The concrete instantiations of these predicates are defined in the ground values (`evidence.db` file). We defined for example, that Anna smokes (`Smokes(Anna)`). We will call instances of x (like $Anna$) objects from now on.

We can also define predicates which connect two ore more objects. The predicate `Friends(person, person)` connects for example two persons x and y with each other.

Types

Types are useful to separate the objects in the input data. In our example there exist only one type $person$. However, we can use as many types as we want. If we think for example of a predicate `teach(person, course, semester)`, we defined the three different types $person$, $courses$, and $semester$. More formally, types form distinct sets of objects have no overlapping instances. Even if the ids of two objects belonging to two different types are equal, these objects have nothing

to do with each other. In our $teach$ relation this is a very useful feature, since there must be no overlapping between $person$, $course$, and $semester$.

Build-in Type float_ Usually, all types represent objects consisting of string values. The build-in type `float_` represents float numbers. We could for instance extend our *Friends* predicate to `Friends(person, person, float_)`, where the last parameter could indicate how much the two persons like each other. Then, the grounding `Friends(Anna, Bob, 0.8)` would mean a stronger friendship than `Friends(Anna, Edward,0.1)`

These numeric values are important for defining sophisticated *soft formulas*. The reader is referred to Section 2.3.1 for further details.

Observed and hidden predicates

Predicates can either be $hidden$ or $observed$. Observed predicates are marked with a $\star$ in front of the predicate definition, while hidden predicates do not have an additional identifier. In our example, there exists one observed predicate `*Friends(person, person)` and two hidden predicates `Smokes(person)`, and `Cancer(person)`.

Observed Predicates: For observed predicates the closed world assumption holds. This means that all ground atoms of this predicate not listed in the evidence are false or, in other words, each possible grounding of an observed predicate must be given due to evidence. The true ground literals are given in the input file `evidence.db`. Observed predicates are not part of the solution, since their ground values do not *change*.

Hidden Predicates: Hidden predicates are predicates, where we do not know their ground values after solving the model (a-posteriori). From a database view, we can also refer to them as query predicates. Their ground values *change* during the MAP state calculation. When we solve the MAP query, we are in principle interested in the a-posteriori groundings of the hidden predicates. Thus, they will also be returned in the solution. In our small example, we want to compute who has cancer (hidden predicate `Cancer(person)`), and who are smokers (hidden predicate `Smokes(person)`). In case of the latter one, we have already some a-priori groundings given, namely that `Smokes(Anna)` and `Smokes(Edward)` must hold in the solution. For the first one, no a-priori groundings are given.

2.2.4 Formulas

With the defined observed and hidden predicates, we can now define formulas (also called rules or restrictions) which describe our domain. A formula consists of at least two predicates connected with logical disjunctions. At least one of the predicates in a formula must be hidden.

The connection between the predicates is done by so called $variables$. Let us inspect for instance the formula `!Smokes(x) v Cancer(x).` Here the variable `x` connects the two predicates `smokes` and `cancer` with each other. In principle, the variable represents all instantiations of the underlying type. Thus, in our example the corresponding first order formula is:

$$\forall x \neg Smokes(x) \vee Cancer(x) \quad x \in persons$$

As a consequence, one variable is assigned to exactly one type. If two different types refer to the same variable, rockIt returns an exception. Each formula opens a new variable namespace. Thus, the variable names can be reused in every formula. There is no connection between them. In our example, the formula `!Friends(x,y) v !Smokes(x) v Smokes(y)` can also use the variable name `x`, which is completely independent from the previous formula.

In the classical Markov logic definition, the only formula type which exists are disjunctions assigned with a weight. We will refer to these formula types as *soft formulas*. In our small example, we used only these types of formulas. Apart from soft formulas, rockIt can also deal with *hard formulas*, which must hold in the certain domain, and with *cardinality constraints*, simplifying for instance the definition of functional one-to-one restrictions.

Section 2.3 explains these three formula types and their different possible applications in more detail.

2.2.5 Variables

Variables are used inside formulas and represent constants. In our example, we have e.g. the variable `x` which represents several potential constants like `Anna`, `Bob`, and so on.

Classical Variables

Classical variables represent a set of constants. In our example above, we have only defined classical variables (like `x` and `y`). They represent all constants determined by its type. The type of a variable is inferred from the respective predicate definition. Different from Tuffy and Alchemy, these variables can also start with a capital letter.

Constants as Variables

We can also use constants as variables. We can for example introduce the following formula

```
0.4 !Friends(x,"Bob") v Smokes(y)
```

which would state that everyone who is friend with "Bob" is more likely to smoke (with weight 0.4).

2.2.6 Ground Values

Ground values represent the a-priori knowledge about predicates. Since there can be a large number of these ground values, they are stored not in the model file, but in a separate one. In our example this file is called `evidence.db`.

The objects within predicate definitions have to be quoted (enclosed by `"`s) if special characters others than small or large letters and − are used. Here are some valid examples:

```
Friends(Anna, "Bob D. Clinton")
Friends(Anna, "Edward Moli\'ere")
Friends(Anna, Frank-Maier)
```

Float Numbers in Ground Values In case a predicate definition uses the build-in type `float_` (see types section above), the objects in the ground values might also be float numbers. Let us assume that we defined the predicate `Friends(person, person, float_)`. Then the following examples are valid:

```
Friends(Anna, "Bob D. Clinton", 1)
Friends(Anna, "Edward Moli\'ere", 0.00120)
Friends(Anna, Frank-Maier, -123.2)
```

It is not possible to use shortcuts for power of 10*x like 3.2E-07.

Negated ground values In rockIt it is also possible to define negated ground values. Negated ground values start with an `!` symbol. Let us extend Listing 2.2 with the negated ground value

```
!Friends(Anna, Garry)
```

This negated ground value will have no effect on the solution. Thereby it is important to know that we assume *closed world assumption*. In this case this means that if we do not know if a statement is true, we assume that it is false. Consequently,

since the predicate `Friends` is observed, the new negated ground axiom has no effect on the solution.

If we, however, add the following negated ground value

```
!Cancer("Bob")
```

we will get a solution where `Bob` does not smoke and has no cancer. The difference is that the predicate `Cancer` is hidden. The negated ground value is a hard fact (that is not allowed to be violated in the final MAP state), while all formulas in our small example are soft formulas[3]. Soft formulas must not hold in the final MAP state. Thus, the negated ground value has a *stronger* effect on the final MAP state.

2.3 Formula Types

In rockIt three different formula types exist. First, we have the so called *soft formulas*, which are the classic Markov logic formulas. Furthermore, rockIt also supports the definition of *hard formulas* and *cardinality formulas*.

2.3.1 Soft Formulas

Soft formulas consist of conjunctions of predicates (refer to Section 2.2.3) assigned with a weight. Intuitively speaking, the higher that weight is, the more likely the formula is true in the respective model. For a in-dept discussion of the semantics of soft formulas, refer to [2].

Listing 2.1 contains the following three examples of classic first order formulas.

```
// Smokes(x) => Cancer(x)
0.5  !Smokes(x) v Cancer(x)

// Friends(x, y) => (Smokes(x) <=> Smokes(y))
0.4  !Friends(x,y) v !Smokes(x) v Smokes(y)
0.4  !Friends(x,y) v Smokes(x) v !Smokes(y)
```

The formulas need to be normalized to disjunctions. The weight can have any value (must not be between 0 and 1) and has to be written before the formula.

Individual Weights for Every Ground Axiom

rockIt allows having soft formulas where each axiom has a separate weight. In Listing 2.3 we illustrate this on a new example. In this example we compute who

[3] We refer the reader to Section 2.3.1 for further information on soft formulas.

is friend of whom (with the hidden predicate `Friends`) given an individual degree of friendship `FriendsDegree`.

Compared to usual soft formulas, we do not provide a concrete weight, but a variable `conf`. This variable points to the floating-point number type of the `FriendsDegree` predicate. Thus, each instantiation of the formula will get the weight of the indicated floating point number. These weights are defined in the ground value file displayed in Listing 2.4. Taking the first entry `FriendsDegree(Anna, Bob, 0.5)` as an example, the weight that Anna and Bob are friends is 0.5.

Listing 2.3: Model Definition of Friends Example

```
// Predicates
// observed predicates
*FriendsDegree(person, person, float_)
// hidden predicates
Friends(person, person)


// Formulas
// conf: FriendsDegree(x,y,conf) => Friends(x,y)
conf: !FriendsDegree(x,y,conf) v Friends(x,y)
```

Listing 2.4: Ground Value Definition of Friends Example

```
FriendsDegree(Anna, Bob, 0.5)
FriendsDegree(Anna, Edward, 0.1)
FriendsDegree(Anna, Frank, 0.8)
FriendsDegree(Edward, Frank, -0.5)
FriendsDegree(Gary, Edward, 0.2)
FriendsDegree(Frank, Edward, 0.3)
```

The resulting MAP state of this small example declares everyone as friend who has a positive `FriendsDegree` weight. Thus, Edward and Frank are no friends.

```
Friends("Anna", "Bob")
Friends("Anna", "Edward")
Friends("Anna", "Frank")
Friends("Gary", "Edward")
Friends("Frank", "Edward")
```

2.3.2 Hard Formulas

Hard formulas are formulas that must hold in the MAP state. In other words, there must be no violation of any hard formula in the solution.

As an example, let us assume that if a person x is a friend of another person y, this person y is also a friend of person x (symmetry of the friends predicate). Further, let us assume that this new formula must always hold and, thus, must be modeled as hard formula. Therefore, the following new hard formula has to be added to our friends example of Listing 2.3.

```
//Friends(x,y) => Friends(y,x)
!Friends(x,y) v Friends(y,x).
```

Since the formula must always hold, we must not provide a weight for this formula. The dot `.` at the end of the formula indicates that this formula is hard.

If we again compute the MAP state of our updated model, we get the following results:

```
Friends("Anna", "Bob")
Friends("Anna", "Edward")
Friends("Anna", "Frank")
Friends("Gary", "Edward")
Friends("Bob", "Anna")
Friends("Edward", "Anna")
Friends("Frank", "Anna")
Friends("Edward", "Gary")
```

2.3.3 Existential Formulas

Apart from universally quantified formulas, we are also able to formulate existential quantification in first-order logic. In rockIt we decided to use a little bit different syntax here, which also allows to define 'at least n' formulas (refer to next Section).

At the moment, we can only define hard existential quantification formulas. Thus, these formulas must hold in the knowledge base.

Simple existential formula Let us assume that we have the first order formula $\forall\, x\, \exists\, y\, Friends(x, y)$

This translates to the following formula in rockIt:

```
//\forall  x  \exists  y Friends(x,y)
|y| Friends(x,y) >= 1
```

You have to surround the existential variable with `|`. The MAP state then returns the following:

```
Friends("Bob", "Bob")
Friends("Anna", "Edward")
Friends("Bob", "Anna")
Friends("Anna", "Bob")
Friends("Anna", "Anna")
Friends("Bob", "Gary")
Friends("Bob", "Frank")
Friends("Anna", "Gary")
Friends("Bob", "Edward")
Friends("Anna", "Frank")
Friends("Edward", "Anna")
Friends("Edward", "Bob")
Friends("Frank", "Gary")
Friends("Edward", "Edward")
Friends("Edward", "Gary")
Friends("Frank", "Anna")
Friends("Frank", "Bob")
Friends("Frank", "Edward")
Friends("Frank", "Frank")
Friends("Gary", "Gary")
Friends("Gary", "Frank")
Friends("Gary", "Anna")
Friends("Gary", "Edward")
Friends("Gary", "Bob")
```

At a first glimpse the results seems to be wrong since very many literals are set to true in the final MAP state. However, since most of them have 0 weight, it is not wrong to return them. To avoid having so many results, we can use a simple trick. We add a small negative weight to each `Friends` literal:

```
-0.0001 Friends(x,y)
```

Then, we obtain the more compact MAP state:

```
Friends("Bob", "Gary")
Friends("Anna", "Edward")
Friends("Anna", "Bob")
Friends("Anna", "Frank")
Friends("Edward", "Anna")
Friends("Frank", "Edward")
Friends("Gary", "Edward")
```

'At least n' Formula We can also add formulas which express that we want to retrieve at least n results of certain variables.

This is equivalent with the first order formula: $\exists^{\geq 7} x, y \; Friends(x, y)$

```
//Returns at least 7 friends
|x,y| Friends(x,y) >= 7
```

To avoid getting a MAP state with many true groundings, we again add the formula

```
-0.0001 Friends(x,y)
```

Then, we obtain the MAP state:

```
Friends("Bob", "Edward")
Friends("Anna", "Edward")
Friends("Anna", "Bob")
Friends("Anna", "Frank")
Friends("Edward", "Bob")
Friends("Frank", "Edward")
Friends("Gary", "Edward")
```

Important: Because we can not apply cutting plane inference in combination with existential formulas (or with 'at least n' formulas) runtimes might be long and models might require large amount of RAM. Cutting plane inference is automatically disabled for these formulas. For all other formulas it remains enabled.

2.3.4 Cardinality Formulas

With cardinality formulas we can model various scenarios including one-to-one, one-to-many, many-to-one, but also two-to-many and so on. Therefore, we need two additional inputs:

1. The cardinality number which is an integer number and follows at the end of the formula after the symbols <= or >=.

2. The over-variables. Each combination of the variables which are NO over-variables occur 'cardinality number' times in the final MAP-state.

The easiest way to understand the over-variables is to provide small examples.

many-to-one If we want to include a many-to-one restriction to our example of Listing 2.3, we have to add the following formula.

```
//Friends is many-to-one
|x| Friends(x,y) <= 1
```

In this case, every object of variable `x` (over-variable) has maximal `1` (cardinality number) other object. Or in other words, every constant in `y` is distinct from each other. The MAP-state now contains the following groundings:

```
Friends("Anna", "Bob")
Friends("Frank", "Edward")
Friends("Anna", "Frank")
```

As we can see from the results, there can be many equal objects of `x` (for instance `Anna`), but the objects of `y` must be distinct.

one-to-many If we want to include a one-to-many restriction to our example of Listing 2.3, we have to add the following formula.

```
//Friends is one-to-many
|y| Friends(x,y) <= 1
```

The MAP-state now contains the following groundings:

```
Friends("Frank", "Edward")
Friends("Gary", "Edward")
Friends("Anna", "Frank")
```

one-to-one The one-to-one restriction is just a combination of the one-to-many and many-to-one restriction. Thus, we have to add the following two formulas:

```
//Friends is many-to-one
|x| Friends(x,y) <= 1
//Friends is one-to-many
|y| Friends(x,y) <= 1
```

The resulting MAP-state is:

```
Friends("Anna", "Frank")
Friends("Frank", "Edward")
```

Top-k The following example returns the top-3 friends. Therefore we have to add the following formula to Listing 2.3.

```
//The top 3 Friends
|x,y| Friends(x,y) <= 3
```

The results are the following:

```
Friends("Anna", "Bob")
Friends("Anna", "Frank")
Friends("Frank", "Edward")
```

More complex cardinality constraints Cardinality constraints can also contain more than one predicate. Since our small example only contains one predicate, we provide a sample formula from another domain. Assume that you would want to restrict the total number of cars and bicycles to 5. Therefore, the following formula is needed:

```
// more complex cardinality restriction
|x,y|  Cars(x) v Bicycles(y) <= 5
```

2.4 Learning

rockIt supports learning in Markov logic by implementing a simple Voted Perceptron using multiple worlds and the network as the input.

2.4.1 Voted Perceptron

The explations in this section are largely based on [1].
Like the majority of the training algorithms for log-linear models, learning formulae weights for Markov logic networks can be realised with standard learning methods based on the gradient of the conditional likelihood function.

Below, you will find the algorithm for the voted perceptron in pseudo code (Algorithm 1). In the current implementation, the learning rate is hard coded as η=1. This may become adjustable in the future. The number of epochs T can be set via the *-iterations* parameter.

Algorithm 1 Voted perceptron algorithm for Markov logic with T epochs

$w_0 \leftarrow 0$
$epochWeight_0 = 0$
for $t \leftarrow 1...T$ **do**
  for $i = 1...N$ **do**
    $y_{MAP} \leftarrow ILP(x, y)$
    $w_i \leftarrow w_{i-1} + \eta[n(y_{CurrentModel}) - n(y_{MAP})]$
  end for
  $epochWeight_t = \frac{1}{N} \sum_{i=1...N} w_i$
end for
return $\frac{1}{T} \sum_{t=1...T} epochWeight_t$

2.4.2 Simple Learning Example

We now give an example of the Voted Perceptron Learning. In this example, we assume that there is some connection between a person being rich and their happiness (or unhappiness). Before learning, this connection is unknown, i.e., if a person is rich, the probability of them being happy is the same as them being not happy. We define our model in the file `prog.mln` such that `rich` is an observed predicate and happy is a `hidden` predicate:

Listing 2.5: Model Definition of Learning Example

```
*rich(person)
happy(person)

      0.0 !rich(person) v happy(person)
```

We create two worlds in the files `evidence1.db` and `evidence2.db`.

1. `evidence1.db`

```
rich(Anna)
rich(Bob)
rich(Charlie)
rich(Dannie)

happy(Anna)
happy(Charlie)
```

2. `evidence2.db`

```
rich(Anna)
rich(Bob)
rich(Charlie)

happy(Anna)
happy(Charlie)
```

In the following, these worlds will be referred to as r_1 and r_2 (r for *reality*, as w are weights). As algorithm 1 shows, in each epoch we look at all the worlds and update the weights. The weights are then updated based on the weight of the previous world w_{i-1} and a learning rate η (in the basic implementation $\eta = 1$) which is applied to the difference between the number of expected positives ($n(y_{CurrentModel})$) and actual positives ($n(y_{MAP})$):

$$w_i \leftarrow w_{i-1} + \eta[n(y_{CurrentModel}) - n(y_{MAP})]$$

First of all, it has to be noted that a weight of zero will always be considered rather negative, i.e., the MAP state will be that the grounding is false. Now we start the learning process.

Epoch 1:
Starting with the weight of 0.0 from the formula, r_1 contains two true groundings (namely that Anna and Charlie are rich and happy), but in the MAP state we would expect none. Hence, the weight is updated:

$$2.0 = 0.0 + 1.0 * (2 - 0)$$

Now we look at r_2 starting with a weight of 2.0 for the learning (which is the weight we just calculated). In r_2, there are two happy people, Anna and Charlie, but given the current weight we would expect to find three. The weight seems to be too high and the update will adjust that:

$$1.0 = 2.0 + 1.0 * (2 - 3)$$

We have look at all worlds in the first epoch and if we were to stop now, this would be the final result. For the remaining epochs, the calculation of the weights will be exactly the same, so we will only give the weight updates and try to explain the results.

Epoch 2:

$$r_1 : -1.0 = 1.0 + 1.0 * (2 - 4)$$
$$r_2 : 1.0 = -1.0 + 1.0 * (2 - 0)$$

Epoch 3:

$$r_1 : -1.0 = 1.0 + 1.0 * (2 - 4)$$
$$r_2 : 1.0 = -1.0 + 1.0 * (2 - 0)$$

For simplicity, we stop here after $T = 3$ epochs. The final weight of the formula is now calculated as the average over all the epoch weights or simply the average over all weights.:

$$w_T = \frac{1}{T} \sum_{t=1}^{T} epochWeight_t$$

where the $epochWeight_t$ is the average over the weights in epoch t.

$$w_3 = \frac{1}{6}(2.0 + 1.0 + (-1.0) + 1.0 + (-1.0) + 1.0) = 0.5$$

It seems only fitting that the result weight is (just) slightly higher than zero as we observed two worlds with rich people that are not necessarily happy. Note that a learning rate $\eta = 1$ is rather high and thus causes the weights within the epochs to somewhat "jump".

2.5 Detailed Input Format

Paste antlr3 Syntax (without Java Code + explanations)

2.6 Configuration Possibilities

There are some possible parameters that can be adjusted in:

The rockit.properties file Please check the `rockIt.properties` file for detailed discussions about the different parameter settings.

The runnable jar Just call `java -jar rockit.jar` to get further informations. For developers: The Main class is located in `com.googlecode.rockIt.app.Main`.

Some parameters can be changed in both, the rockIt.properties file and in the jar execution.

Chapter 3

Comparison with other existing Markov logic solvers

Besides rockIt there exist three other Markov logic solvers.

First, there is the oldest and most established system **Alchemy**. The advantage of alchemy is that there exists the largest community discussing about problems of alchemy. However, there are several disadvantages. The performance of alchemy is very bad since it does not use databases. Consequently, the grounding takes very long and even for medium sized models the execution time might be very large. Furthermore, hard formulas in alchemy are not guaranteed to hold in the final MAP state. This is due to the fact that Alchemy only sets very large weights for hard formulas which may be violated in the MAP state.

Second, there exist the **Tuffy** System. The tuffy system is faster in performance than the Alchemy system. However, our experiments show that rockIt is even faster than Tuffy. Refer to our AAAI publication (link see at the end of this document) for details. The main disadvantage of Tuffy is the handling of negative weights in formulas. Let us assume Tuffy gets the following formula as an input, where $student$ is an observed predicate and $advisedBy$ is a hidden predicate:

$$\langle student(a1) \vee \neg advisedBy(a2, a1) \vee \neg advisedBy(a1, a2), -2 \rangle .$$

Since Tuffy can not deal with negative weights, Tuffy transforms this to the following formula with positive weight:

$$\langle student(a1) \wedge advisedBy(a2, a1) \wedge advisedBy(a1, a2), 2 \rangle .$$

However, Tuffy can not handle disjunctions. Consequently this formula is again transformed as followed:

$$\langle student(a1) \vee advisedBy(a2, a1), 1 \rangle .$$

$$\langle student(a1) \vee advisedBy(a1, a2), 1\rangle \,.$$

As illustrated the resulting two equations have rarely to do with the original one. In order to be able to compare the results and the runtimes of rockIt with the one of Tuffy, we have the parameter `simplify_negative_weight_and_conjunction` (refer to the `rockIt.properties` file for changing the parameter and for further information), which enables such transformations. If disabled, negative weights are treated correctly.

Another disadvantage of the Walk-Sat algorithm used by Tuffy and Alchemy is the unknown gap that the algorithm produces. The user of the systems do never know how far the actual MAP-state solution differs from the optimal solution. Since we leverage integer linear programming techniques to determine the MAP-state we can exactly ensure that our solution is correct up to a certain gap.

The third markov logic system is TheBeast[1]. This system also does not violate hard formulas, which is an advantage. Still, the performance is slower than the one of rockIt and the input syntax differs from Alchemy, Tuffy, and rockIt. Furthermore, the further development is closed down.

As a conclusion, the following advantages of rockIt envolve:

- **rockIt is the fastest** of all existing Markov logic solvers (as far as we know).
- **rockIt uses cutting plane aggregation** which further speeds up the solution process.
- **rockIt handels hard formulas as deterministic knowledge**. The MAP state will never violate any hard formula.
- **rockIt is able to correctly interpret negative weights**.
- **rockIt gives the possibility to decide about the exactness of the MAP state solution** with the `gap` parameter. Other systems which are based on the Walk-Sat algorithm like Tuffy and Alchemy can not provide a parameter to configure how far their MAP state differs from the optimal MAP state.
- **rockIt uses the standard Markov logic syntax**. We tried to stay as close as possible to the standard Markov logic syntax developed by Alchemy.

Since the recent development of rockIt, it still has the following missing features:

- rockIt does not yet support weight learning.

The implementation of these features is straight forward and has not been done due to time constraints.

[1] http://code.google.com/p/thebeast/

Chapter 4

Feedback

If you use rockIt for your project and it has been helpful (or not) we would be happy to hear about it. If you use rockIt in your research, please cite us! We would also be happy if you could send us a copy of any published work that uses rockIt. If you find an error or if you have ideas for improvements it would be very helpful if you could write a ticket at our issue tracker `http://code.google.com/p/rockIt/issues/list`.

Jan Noessner, Mathias Niepert and Heiner Stuckenschmidt. **rockIt: Exploiting Parallelism and Symmetry for MAP Inference in Statistical Relational Models**. In: *Main track of the Twenty-Seventh Conference on Artificial Intelligence (AAAI 2013)* (upcoming). July 1418, 2013 in Bellevue, Washington, USA, 2013. `http://arxiv.org/abs/1304.4379`

Bibliography

[1] Rim Helaoui. TODO TODO TODO TODO TODO.

[2] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1):107–136, 2006.