OpenGM 2.0 Manual

Users' and Developers' Section

OpenGM 2.0 was developed jointly by
Bjoern Andres[1], **Thorsten Beier**[2] and **Jörg H. Kappes**[3]

at the
Heidelberg Collaboratory for Image Processing
University of Heidelberg (Germany)

This manual is written for researchers and technical practitioners who are familiar with the very basics of discrete graphical models and have some experience in generic programming in C++, e.g. from using the Standard Template Library (STL).

OpenGM	hci.iwr.uni-heidelberg.de/opengm2
CMake	http://www.cmake.org/
HDF5	http://www.hdfgroup.org/
Boost	http://www.boost.org/
Doxygen	http://www.stack.nl/~dimitri/doxygen/

[1] bandres@seas.harvard.edu
[2] thorsten.beier@iwr.uni-heidelberg.de
[3] kappes@math.uni-heidelberg.de

Contents

Chapter 1

Getting Started

1.1 What is OpenGM?

OpenGM is a C++ template library for discrete factor graph models and distributive operations on these models. It includes state-of-the-art optimization and inference algorithms[1] beyond message passing. OpenGM handles large models efficiently, since (i) functions that occur repeatedly need to be stored only once and (ii) when functions require different parametric or non-parametric encodings, multiple encodings can be used alongside each other, in the same model, using included and custom C++ code. No restrictions are imposed on the factor graph or the operations of the model. OpenGM is modular and extendible. Elementary data types can be chosen to maximize efficiency. The graphical model data structure, inference algorithms and different encodings of functions interoperate through well-defined interfaces. The binary OpenGM file format is based on the HDF5 standard and incorporates user extensions automatically.

1.2 Compiling and Installing OpenGM

OpenGM is a header-only C++ template library and can be used as such by simply decompressing the archive in a desired location. However, OpenGM also comes with command line tools, tutorials and unit tests that can be built using Cross Platform Make (CMake). We recommend to use the Cmake GUI and proceed as follows:

- Select the source directory.
- Select a build directory that is different form the source directory.
- Activate the check boxes labeled "Grouped" and "Advanced" for a comprehensive and structured overview of options.
- Select from the group BUILD optional components of OpenGM, e.g. BUILD_TESTING, BUILD_COMMANDLINE.
- Select from the group WITH optional external libraries, e.g. WITH_BOOST , WITH_CPLEX. Any such libraries need to be installed as described in Section 1.3.
- Click the button labeled "Configure" and follow the instructions.
- Click the button labeled "Generate" and follow the instructions.

[1] Not all can be used for each semiring.

1.2.1 Linux

On a Linux operating system, CMake writes make files such that OpenGM can now be built from the command line, in the build directory, by simply entering

```
> make
```

Unit tests are built if the CMake option BUILD_TESTING is set. To run these tests and verify that OpenGM is working properly, enter

```
> make test
```

To re-build the Doxygen reference documentation from the source code, enter

```
> make doc
```

To install OpenGM in a directory that can be specified using CMake, enter

```
> make install
```

1.2.2 Windows

On a Windows system with Microsoft Visual C++ installed, CMake can generate a Microsoft Visual Studio solution file that contains, among others, a target called ALL_BUILD. To build all components of OpenGM selected through CMake, open the solution file in Visual Studio, right click on the target ALL_BUILD and hit "Build".

1.3 Installing Optional Third Party Libraries

OpenGM is a header-only C++ template library that depends exclusively on the C++ standard template library and can therefore be used without installing any third party code.

There are, however, optional components of OpenGM that are built on 3rd party code. In particular, OpenGM provides wrappers around third party inference code which make it possible to use this code with models that have been built with OpenGM. Third party code and binaries are published under different, possibly more restrictive licenses than OpenGM, cf. Section 8.

For HDF5, Boost, IBM ILOG CPlex, ConicBundle and LibDAI, it is sufficient to download a copy and add the required path to the CMake environment. MaxFlow, IBFS, QPBO, TRW-S and MRF-LIB need to be patched in order to resolve naming conflicts. For Linux, we provide a shell script that downloads the respective 3rd party code to the directory external within the OpenGM source tree and applies the necessary patches. Code that has been installed this way will be detected automatically by CMake. This script can be invoked from the command line:

```
> make externalLibs
```

Windows users can either use Cygwin[2] to run the Linux script or download the libraries and apply the patches manually. Any feature of OpenGM that depends on 3rd party code needs to be activated by setting the corresponding CMake environment variable in the group WITH as described in Section 1.2. Here is a complete list of 3rd party libraries that enable additional features of OpenGM. Those marked with a * need to be patched:

[2] http://www.cygwin.com

- **HDF5** [12] HDF5 is a data model, library, and file format for storing and managing data which is used in OpenGM so save graphical models and result files of the command line tool.
- **Boost** [16] The Boost graph library is used for calculation of MaxFlow in graphcut-based methods.
- **CPLEX** [15] CPlex is a commercial library for linear, quadratic and integer programming. For academics a free version exists. In OpenGM Cplex is used for LP relaxations and ILP formulations.
- **ConicBundle** [14] The Conic Bundle Library is published under the GPL2 and used for dual decomposition bundle-method.
- **MaxFlow*** [21] The max flow implementation of Vladimir Komogorov can be used in graphcut-based methods.
- **IBFS*** [13] The max flow implementation by Incremental Breadth-First Search of Sagi Hed can be used in graphcut-based methods.
- **QPBO*** [22] OpenGM provides a wrapper to the original QPBO-code of Vladimir Komogorov.
- **TRWS*** [23] OpenGM provides a wrapper to the original TRWS-code of Vladimir Komogorov with the option to copy the data into their own data structure or using a view.
- **MRF-LIB*** [33] OpenGM provides a wrapper to the original MRF-Library with the option to copy the data into their own data structure or using a view.
- **libDAI** [28] OpenGM provides a wrapper to lib-DAI, but copy the data into their own data structure.

Chapter 2

Graphical Models

2.1 Mathematical Foundation

OpenGM is built on a rigorous definition of the syntax and semantics of a graphical model. The syntax determines a class of functions that factorize w.r.t. an associative and commutative operation. In a probabilistic model, it determines the conditional independence assumptions. The semantics specify the operation and one function out of the class of all function that are consistent with the syntax.

The *syntax* (Fig. 2.1a) consists of a factor graph, i.e. a bipartite graph (V, F, E), a linear order $<$ in V, a set I whose elements are called *function identifiers*, and a mapping $\gamma : F \to I$ that assigns one function identifier to each factor such that only factors that are connected to the same number of variables can be mapped to the same function identifier.

For any $v \in V$ and $f \in F$, the *factor* f is said to *depend* on the *variable* v iff $(v, f) \in E$. $\mathcal{N}(f)$ denotes the set of all variables on which f depends and $(v_j^{(f)})_{j\in\{1,\dots,|\mathcal{N}(f)|\}}$ the sequence of these variables in ascending order. Similarly, $(v_j)_{j\in\{1,\dots,|V|\}}$ denotes the sequence of *all* variables in ascending order.

Semantics (Fig. 2.1b) w.r.t. a given syntax consist of one finite set $X_v \neq \emptyset$ for each $v \in V$, a commutative monoid $(\Omega, \odot, 1)$ and for any $i \in I$ for which there exists an $f \in F$ with $\gamma(f) = i$, one function[1] $\varphi_i : X_{v_1^{(f)}} \times \cdots \times X_{v_{|\mathcal{N}(f)|}^{(f)}} \to \Omega$.

The function from $X := X_{v_1} \times \cdots \times X_{v_{|V|}}$ to Ω *induced* by syntax and semantics is the function $\varphi : X \to \Omega$ such that $\forall (x_{v_1}, \dots, x_{v_{|V|}}) \in X$:

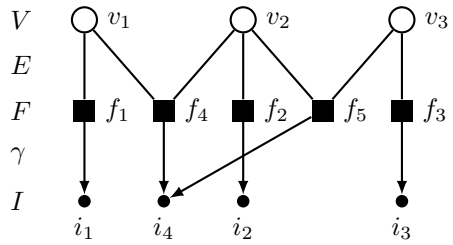
$$\varphi(x_{v_1}, \dots, x_{v_{|V|}}) := \bigodot_{f\in F} \varphi_{\gamma(f)} \left(x_{v_1^{(f)}}, \dots, x_{v_{|\mathcal{N}(f)|}^{(f)}} \right) \quad . \tag{2.1}$$

W.l.o.g., OpenGM simplifies the syntax and semantics by substituting $V = \{0, \dots, |V| - 1\}$, equipped with the natural order, $F = \{0, \dots, |F| - 1\}$, $I = \{0, \dots, |I| - 1\}$ and for each $v \in V$, $X_v = \{0, \dots, |X_v| - 1\}$. A graphical model is thus completely defined by the number of variables $|V|$, the number of labels $|X_v|$ of each variable $v \in V$, the edges E of the factor graph, the number of functions $|I|$, the assignment of functions to factors γ, the commutative monoid $(\Omega, \odot, 1)$ and one function φ_i for each function identifier $i \in I$.

2.2 The Graphical Model Class Template

The four parameters of the class template GraphicalModel define 1. the co-domain of the function, i.e. the type of values the function can attain, 2. the operation w.r.t. which the objective function factorizes (Section 2.5), 3. data types to encode functions (Section 2.4), and 4. a label space that

[1] The existence of φ implies $\forall f, f' \in F : \gamma(f) = \gamma(f') \Rightarrow \forall j \in \{1, \dots, \deg(f)\} : X_{v_j^{(f)}} = X_{v_j^{(f')}}$.



a. Syntax

$$X_{v_1} = X_{v_2} = X_{v_3} = \{0, \dots, n\} =: X_v$$

$$\varphi_{i_1}, \varphi_{i_2}, \varphi_{i_3} : X_v \to \Omega, \quad \varphi_{i_4} : X_v^2 \to \Omega$$

$$\varphi : X_v^3 \to \Omega \text{ such that } \forall (x_{v_1}, x_{v_2}, x_{v_3}) \in X_v^3:$$
$$\varphi(x_{v_1}, x_{v_2}, x_{v_3}) = \varphi_{i_1}(x_{v_1})\varphi_{i_2}(x_{v_2})\varphi_{i_3}(x_{v_3}) \cdot$$
$$\cdot \varphi_{i_4}(x_{v_1}, x_{v_2})\varphi_{i_4}(x_{v_2}, x_{v_3})$$

b. Semantics

Figure 2.1: A factor graph (V, F, E) describes how a function φ decomposes into a product of functions. In OpenGM, we extend this syntax by a set I of *function identifiers* and a mapping $\gamma : F \to I$ that assigns one function identifier to each factor. In the above example, the factors f_4 and f_5 are mapped to the same function identifier i_4, indicating that the corresponding functions $\varphi_{i_4}(x_{v_1}, x_{v_2})$ and $\varphi_{i_4}(x_{v_2}, x_{v_3})$ are identical.

defines a. the domain of the function, and b. the data types to index elements (variables and labels) in this domain (Section 2.3). A graphical model type is always defined like this:

```
typedef opengm::GraphicalModel<ValueType, OperationType, FunctionTypeList, SpaceType> Model;
```

Suitable classes for each parameter are discussed in detail in the following sections. Here are three concrete examples: First, a sum of double-valued functions which are stored explicitly, i.e. as value tables:

```
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/graphicalmodel/space/discretespace.hxx>
#include <opengm/functions/explicit_function.hxx>
#include <opengm/operations/adder.hxx>

typedef opengm::GraphicalModel<
    double,
    opengm::Adder,
    opengm::ExplicitFunction<double, size_t, size_t>,
    opengm::DiscreteSpace<size_t, size_t>
> Model;
```

Second, a product of functions:

```
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/graphicalmodel/space/discretespace.hxx>
#include <opengm/functions/explicit_function.hxx>
#include <opengm/operations/multiplier.hxx>

typedef opengm::GraphicalModel<
    double,
    opengm::Multiplier,
    opengm::ExplicitFunction<double>,
    opengm::DiscreteSpace<>
> Model;
```

And third, a model in which two types of functions, explicit functions and sparse functions, can be used alongside each other[2]:

[2]The class template opengm::meta::TypeListGenerator can assemble lists of up to 16 types. Type lists with more entries can be constructed using opengm::meta::TypeList in opengm/utilities/metaprogramming.hxx

```
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/graphicalmodel/space/discretespace.hxx>
#include <opengm/functions/explicit_function.hxx>
#include <opengm/functions/sparsemarray.hxx>
#include <opengm/operations/multiplier.hxx>
#include <opengm/utilities/metaprogramming.hxx>

typedef opengm::meta::TypeListGenerator<
    opengm::ExplicitFunction<double>,
    opengm::SparseMarray<double>
>::type FunctionTypeList;

typedef opengm::GraphicalModel<
    double,
    opengm::Multiplier,
    FunctionTypeList,
    opengm::DiscreteSpace<>
> Model;
```

2.3 Label Spaces

The first step towards constructing (instantiating) a graphical model in OpenGM is to define a number of variables and, for each variable, the number of labels this variable can attain[3]. This is done by constructing an object called a label space.

In a space with n variables, these variables are indexed by integers ranging from 0 to $n-1$ as is the standard in C++. For each variable j, the labels this variable can attain are indexed in the same way, by integers from 0 to $m_j - 1$ where m_j is the number of labels variable j can attain. All label spaces implement at least the following interface[4].

```
template<class SPACE, class I = std::size_t, class L = std::size_t>
class SpaceBase {
public:
   typedef I IndexType;
   typedef L LabelType;

   IndexType numberOfVariables() const;
   IndexType numberOfLabels(const IndexType) const;
   bool isSimpleSpace() const;

   IndexType addVariable(const IndexType);
};
```

2.3.1 Simple Discrete Space

In the simplest case, all variables have the same number of labels. The class template that handles this case efficiently is SimpleDiscreteSpace. A space with 3 variables each of which can take 4 labels is constructed like this:

```
#include <opengm/graphicalmodel/space/simplediscretespace.hxx>

opengm::SimpleDiscreteSpace<> space(3, 4);
```

2.3.2 Discrete Space

The most general case in which each variable can attain a different number of labels is covered by the class template DiscreteSpace. There are two convenient ways to construct such a space. The

[3]Variables that attain infinitely many labels and graphical models with infinitely many variables are currently not representable in OpenGM.

[4]Space class templates derive from SpaceBase via the curiously recurring template pattern.

first is via indices:

```
#include <opengm/graphicalmodel/space/discretespace.hxx>
opengm::DiscreteSpace<> space;
space.addVariable(4);
space.addVariable(2);
space.addVariable(3);
```

The second is via a pair of iterators that point to the beginning and end of a sequence of label counts:

```
#include <opengm/graphicalmodel/space/discretespace.hxx>
size_t numbersOfLabels[] = {4, 2, 3};
opengm::DiscreteSpace<> space(numbersOfLabels, numbersOfLabels + 3);
```

Both examples construct a discrete space with three variables attaining 4, 2 and 3 different labels, respectively.

2.3.3 Further Reading

More on label spaces can be found in Section 6.1, in the reference documentation and in the header files located in inclue/opengm/graphicalmodel/space.

2.4 Functions

	Closed form $f(x_C) =$	Number of parameters	I/O Support
Explicit Function	θ_{C,x_C}	$\prod_{c\in C} \lvert X_c\rvert$	yes
Sparse Function	$\theta(\xi(x_C))$	$\leq \prod_{c\in C} \lvert X_c\rvert$	yes
Potts Function	$\alpha \cdot (x_a = x_b) + \beta$	2	yes
Generalized Potts Functions	$\alpha(\gamma(x_c))$	Bell number of $\lvert C\rvert$	yes
Multidimensional Potts Function	$\alpha \cdot \prod_{a,b\in C}(x_a = x_b) + \beta$	2	yes
Absolute Label Difference	$\alpha\lvert x_a - x_b\rvert$	1	yes
Squared Label Difference	$\alpha(x_a - x_b)^2$	1	yes
Truncated Absolute Label Difference	$\alpha \max(\lvert x_a - x_b\rvert, T)$	2	yes
Truncated Squared Label Difference	$\alpha \max((x_a - x_b)^2, T)$	2	yes
View Function	$\alpha \cdot gm \rightarrow g(x_C)$	2	no
View Function with Offset	$\alpha \cdot gm \rightarrow g(x_C) + \theta_{x_C}$	$2 + \prod_{c\in C} \lvert X_c\rvert$	no
Memory View Function	$\rightarrow g(x_C)$	1	no

Table 2.1: Types (classes) of functions included in OpenGM

This section describes the types (classes) of functions included in OpenGM (Tab. 2.1) and explains how functions are added to a graphical model. Once a function has been added, it can be used as often as required, with different subsets of variables, as described in Section 2.6.

All functions have at least three template parameters, ValueType, IndexType and LabelType. These need to be consistent with the ValueType of the graphical model class template and with the IndexType and LabelType of the label space, respectively. Both IndexType and LabelType are size_t by default therefore need not be specified.

The general interface to evaluate functions is via operator(). This operator expects one argument, an iterator to the beginning of a sequence of labels, one for each input variable of the function. The main advantage of this function interface is that it easily extends to higher order functions:

```
Function<double> f(...); // constructor depends on function type

std::vector<size_t> labels(2, 0);
double v = f(labels.begin()); // set v = f(0, 0)

Function<double> g(...);
size_t labels2[] = {1, 2, 6, 3};
double w = f(labels2); // set w = f(1, 2, 6, 3)
```

2.4.1 Explicit Function

To describe a function in terms of its full value table, OpenGM provides the class template ExplicitFunction that is based on an efficient implementation of runtime-flexible multi-dimensional arrays [2]. As an example, consider a function on two (so far unspecified) variables each of which assumes four different labels:

```
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/graphicalmodel/space/simplediscretespace.hxx>
#include <opengm/functions/explicit_function.hxx>

typedef opengm::SimpleDiscreteSpace<> Space;
typedef opengm::GraphicalModel<double, opengm::Adder, opengm::ExplicitFunction<double>, Space> Model;

int main() {
    Space space(10, 4);
    Model gm(space);

    size_t shape[] = {4, 4};
    opengm::ExplicitFunction<double> f(shape, shape + 2, 1.0);
    f(0, 0) = 0.2;
    f(1, 1) = 0.2;
    f(2, 2) = 0.2;
    f(3, 3) = 0.2;
    Model::FunctionIdentifier fid = gm.addFunction(f);

    return 0;
}
```

Line 11 constructs the function and initializes all entries in the value table with 1.0. Lines 12–15 change four of the entries. Line 16 adds the function to the graphical model where it can hence be referred to by the function identifier fid.

2.4.2 Sparse Function

A full value table is not a parsimonious description of the function from the previous section: Among the $4 \cdot 4 = 16$ entries in the table, 12 are identical (1.0). Therefore, it makes sense to take 1.0 to be the default value and specify only those entries that differ. The class template SparseMarray is designed for this purpose. The construction of a sparse function is similar to that of explicit functions, only the storage is more efficient:

```
#include <map>

#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/graphicalmodel/space/simplediscretespace.hxx>
#include <opengm/functions/sparsemarray.hxx>

int main() {
    typedef opengm::SimpleDiscreteSpace<> Space;
    typedef opengm::SparseMarray<std::map<size_t, double> > SparseFunction;
    typedef opengm::GraphicalModel<double, opengm::Adder, SparseFunction, Space> Model;

    Space space(10, 4);
```

```
    Model gm(space);

    size_t shape[] = {4, 4};
    SparseFunction f(shape, shape + 2, 1.0);
    f(0, 0) = 0.2;
    f(1, 1) = 0.2;
    f(2, 2) = 0.2;
    f(3, 3) = 0.2;
    Model::FunctionIdentifier fid = gm.addFunction(f);

    return 0;
}
```

2.4.3 Potts Function

Even a sparse function is not the most efficient way to encode the function from the previous section. In fact, the function assumes only two different values, 0.2, if the input variables take on the same label, and 1.0, otherwise. It is a PottsFunction that can, alternatively, be constructed like this:

```
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/graphicalmodel/space/simplediscretespace.hxx>
#include <opengm/functions/potts.hxx>

typedef opengm::SimpleDiscreteSpace<> Space;
typedef opengm::GraphicalModel<double, opengm::Adder, opengm::PottsFunction<double>, Space> Model;

int main() {
    Space space(10, 4);
    Model gm(space);

    opengm::PottsFunction<double> f(4, 4, 0.2, 1.0);
    Model::FunctionIdentifier fid = gm.addFunction(f);

    return 0;
}
```

2.4.4 Potts-N Function

A PottsNFunction assumes one of two values, depending on whether or not *all* variables are assigned the same label. Unlike the classical Potts function, it is also defined for function of more than two variables. However, it should not be confused with the so-called P^N Potts function that is used e.g. in [20]. A PottsNFunction is constructed as follows:

```
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/graphicalmodel/space/simplediscretespace.hxx>
#include <opengm/functions/pottsn.hxx>

typedef opengm::SimpleDiscreteSpace<> Space;
typedef opengm::GraphicalModel<double, opengm::Adder, opengm::PottsNFunction<double>, Space> Model;

int main() {
    Space space(10, 4);
    Model gm(space);

    const double allEqual = 0.2;
    const double notAllEqual = 1.0;

    size_t numbersOfLabels[] = {4, 5, 4};
    opengm::PottsNFunction<double> f(
        numbersOfLabels, numbersOfLabels + 3,
        allEqual, notAllEqual
    );
    Model::FunctionIdentifier fid = gm.addFunction(f);
```

```
    return 0;
}
```

2.4.5 Generalized Potts Functions

A PottsGFunction is invariant under all permutations of labels such that, e.g. $f(1,1,3,4) = f(3,3,4,1)$. Its purpose is to assign different values to different partitions of the set of input variables, regardless which labels are used to describe these partitions. It generalize both the PottsFunction and the PottsNFunction but requires more memory in general. As an example, consider a function that attains the value 10 if all labels are different, 11 if only the first and second are equal, 12 if only the first and third are equal, 13 if only the second and third are equal, and 14 if all are equal:

```
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/graphicalmodel/space/simplediscretespace.hxx>
#include <opengm/functions/pottsg.hxx>

typedef opengm::SimpleDiscreteSpace<> Space;
typedef opengm::GraphicalModel<double, opengm::Adder, opengm::PottsGFunction<double>, Space> Model;

int main() {
    const double allEqual = 0.2;
    const double notAllEqual = 1.0;

    Space space(10, 4);
    Model gm(space);

    size_t numbersOfLabels[] = {4, 4, 4};
    double values[5] = {10, 11, 12, 13, 14};
    opengm::PottsGFunction<double> f(numbersOfLabels, numbersOfLabels + 3, values);
    Model::FunctionIdentifier fid = gm.addFunction(f);

    return 0;
}
```

2.4.6 Absolute Label Difference

AbsoluteDifferenceFunction, SquaredDifferenceFunction, TruncatedAbsoluteDifferenceFunction, and TruncatedSquaredDifferenceFunction are class templates that efficiently implement bi-variate functions that are used, for instance, in computer vision [34].

The absolute difference is a second order function that computes the weighted absolute difference between two labels, multiplied by a weight. The constructor takes the number of labels of the two variables and the weight:

```
#include <opengm/functions/absolute_difference.hxx>
opengm::AbsoluteDifferenceFunction<float> f(4, 4, 0.5);
```

2.4.7 Squared Label Difference

The squared difference is a second order function that computes the squared difference between two labels, multiplied by a weight. The constructor takes the number of labels for the two variables and the weight:

```
#include <opengm/functions/squared_difference.hxx>
opengm::SquaredDifferenceFunction<float> f(4, 6, 0.5);
```

2.4.8 Truncated Absolute Label Difference

The truncated absolute difference is a second order function that computes the truncated absolute difference between two labels and multiplies the result by a weight. If the absolute difference is greater than a threshold, the product if the threshold and the weight is returned. The constructor takes the number of labels of the two variables, the threshold and the weight:

```
#include <opengm/functions/truncated_absolute_difference.hxx>
const float threshold = 20;
const float weight = 0.5;
opengm::TruncatedSquaredDifferenceFunction<float> f(10, 10, threshold, weight);
```

2.4.9 Truncated Squared Label Difference

The truncated squared difference is a second order function that computes the truncated squared difference between the labels, multiplied by a weight. If the squared difference between the labels is greater than a threshold, the product of the threshold and the weight is returned. The constructor takes the number of labels of the two variables, the threshold and the weight:

```
#include <opengm/functions/truncated_squared_difference.hxx>
const float threshold = 20;
const float weight = 0.5;
opengm::TruncatedSquaredDifferenceFunction<float> f(10, 10, threshold, weight);
```

2.4.10 View Functions

The class template ViewFunction is used to treat a subset of one graphical model as a function in another graphical model, without copying data. ViewConvertFunction takes this idea one step further by allowing for arbitrary transformations of the values of the underlying graphical model which are computed only when the view function is evaluated, i.e. in a lazy fashion. ModelViewFunction provides memory efficient scaled views on factors, together with a piecewise offset. It facilitates efficient implementations of algorithms that depend on decompositions of a graphical model, in particular Dual Decomposition [19, 25].

View functions are used internally in inference algorithms and the developer is encouraged to use these. For modeling they are negligible.

2.4.11 Further Reading

More on functions can be found in the reference documentation as well as in the header files located in inclue/opengm/functions.

2.5 Operations

Commutative and associative operations are classes in OpenGM, equipped with member functions for the operation itself and its inverse, the neutral and inverse neutral element, and possibly an (inverse) hyper-operation and order associated with the operation (Table 2.2).

2.5.1 Further Reading

The interface is described in detail in Section 6.3 and in the reference documentation. The source code of the operation classes can be found in include/opengm/operations.

	Operation	Inverse Operation	Neutral Element	Inverse Neutral Element	Hyper-Operation	Inverse Hyper-Operation	Order
opengm::Adder	$+$	$-$	0	∞	$\cdot$	$/$	*none*
opengm::Multiplier	$\cdot$	$/$	1	0	pow	root	*none*
opengm::Maximizer	max	min	$-\infty$	∞	*none*	*none*	$\geq$
opengm::Minimizer	min	max	∞	$-\infty$	*none*	*none*	$\leq$

Table 2.2: Common operations in OpenGM are Adder, Multiplier, Maximizer and Minimizer. Listed above are the corresponding inverse operations, hyper-operations and assoicated orders and neutral elements. Not all operations implement the full interface.

2.6 Factors

Factors connect functions with subsets of variables. Consider the following example, a to be modeled function f that factorizes into five factors which can be described in terms of four functions $f_0, \ldots, f_3$:

$$f(x_0, x_1, x_2) = f_0(x_0)\ f_1(x_1)\ f_2(x_2)\ f_3(x_0, x_1)\ f_3(x_1, x_2) \tag{2.2}$$

Assume that these functions have already been constructed and added to a graphical model as described in Section 2.4, and can now be referenced by the function identifiers fid0, ..., fid3. All that is missing to model (2.2) is to introduce factors:

```
{ size_t vi = 0; gm.addFactor(fid0, &vi, &vi + 1); }                    1
{ size_t vi = 1; gm.addFactor(fid1, &vi, &vi + 1); }                    2
{ size_t vi = 2; gm.addFactor(fid2, &vi, &vi + 1); }                    3
{ size_t vi[] = {0, 1}; gm.addFactor(fid3, vi, vi + 2); }               4
{ size_t vi[] = {1, 2}; gm.addFactor(fid3, vi, vi + 2); }               5
```

When a factor is added it obtains a unique index that is returned by the function addFactor. This index can be used later to access the factor via the index operator of opengm::GraphicalModel:

```
const FactorType& operator[](const IndexType) const;                   1
```

This operator returns a reference to a wrapper class template opengm::Factor. This wrapper class has a convenient interface for accessing the factor's properties. Most importantly, it is identical for all factors, regardless of the function class that is used to encode the factor.

```
template<class GRAPHICAL_MODEL>                                         1
class Factor {                                                          2
public:                                                                 3
   typedef GRAPHICAL_MODEL GraphicalModelType;                          4
   typedef typename GraphicalModelType::ValueType ValueType;            5
   typedef typename GraphicalModelType::LabelType LabelType;            6
   typedef typename GraphicalModelType::IndexType IndexType;            7
   typedef typename std::vector<IndexType>::const_iterator VariablesIteratorType;  8
```

```
    IndexType numberOfVariables() const;
    LabelType numberOfStates(const IndexType) const;
    LabelType shape(const IndexType) const;
    IndexType variableIndex(const IndexType) const;
    ShapeIteratorType shapeBegin() const;
    ShapeIteratorType shapeEnd() const;
    VariablesIteratorType variableIndexBegin() const;
    VariablesIteratorType variableIndexEnd() const;
    template<class ITERATOR>
        ValueType operator()(ITERATOR) const;
}
```

The wrapper class Factor is the main interface to analyze the semantics of a graphical model. It is used extensively by the algorithms described in Sections 3 and 6.4. In fact, it is the key design feature of OpenGM that allows different efficient implementations of functions to be used alongside each other while maintaining a clear separation between graphical models and algorithms that operate on these.

2.6.1 Independent Factors

A factor is strongly coupled to a graphical model. In fact, an instance of opengm::Factor cannot exist without a graphical model because the underlying function is stored only once, within the graphical model. However, it is sometimes useful to consider a factor (i.e. a function along with the indices of variables to which it is bound), independently. The class in OpenGM to facilitate this is called opengm::IndependetFactor. When an independent factor is constructed, usually by casting an opengm::Factor obtained from a graphical model by means of operator[] into an opengm::IndependetFactor, a copy of the function and the variable indices is made in the members of the opengm::IndependetFactor object.

2.6.2 Factor Arithmetic

Independent factors are amenable to factor arithmetic:

```
#include <vector>

#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/graphicalmodel/space/discretespace.hxx>
#include <opengm/functions/explicit_function.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>

typedef opengm::DiscreteSpace<> Space;
typedef opengm::ExplicitFunction<double> Function;
typedef opengm::GraphicalModel<double, opengm::Adder, Function, Space> Model;
typedef Model::FunctionIdentifier FunctionIdentifier;
typedef Model::IndependentFactorType IndependentFactor;

int main() {
    // build model
    size_t variableIndices[] = {3, 4, 5, 4};
    Space space(variableIndices, variableIndices + 8);
    Model gm(space);

    // add functions
    FunctionIdentifier f0, f1;
    { size_t shape[] = {3, 4, 5}; Function f(shape, shape + 3); f0 = gm.addFunction(f); }
    { size_t shape[] = {4, 5, 4}; Function f(shape, shape + 3); f1 = gm.addFunction(f); }

    // add factors
    { size_t vi[] = {0, 1, 2}; gm.addFactor(f0, vi, vi + 3); }
    { size_t vi[] = {1, 2, 3}; gm.addFactor(f1, vi, vi + 3); }

    // simple factor arithmetic
    IndependentFactor a = gm[0] + gm[1];
```

```
    a += 2.0;
    a += gm[0];
    opengm::Adder::op(gm[1], a); // in−place
    IndependentFactor b;
    opengm::Adder::op(gm[1], a, b); // not in−place

    // accumulate over all variables
    double minVal, maxVal;
    std::vector<size_t> argMin;
    a.accumulate<opengm::Minimizer>(minVal);
    a.accumulate<opengm::Minimizer>(minVal, argMin);

    // accumulate over subsets of variables (marginalize)
    IndependentFactor marginal;
    size_t accumulationIndices[] = {1, 2};
    a.accumulate<opengm::Minimizer>(accumulationIndices, accumulationIndices + 2, marginal);

    return 0;
}
```

2.7 Example: A Potts Model

The following working example defines a Potts model on a two-dimensional grid.

```
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/graphicalmodel/space/simplediscretespace.hxx>
#include <opengm/functions/potts.hxx>
#include <opengm/operations/adder.hxx>

using namespace std; // 'using' is used only in example code
using namespace opengm;

const size_t nx = 30; // width of the grid
const size_t ny = 30; // height of the grid
const size_t numberOfLabels = 5;
double lambda = 0.1; // coupling strength of the Potts model

// this function maps a node (x, y) in the grid to a unique variable index
inline size_t variableIndex(const size_t x, const size_t y) {
   return x + nx * y;
}

int main() {
   typedef SimpleDiscreteSpace<size_t, size_t> Space;
   Space space(nx * ny, numberOfLabels);

   typedef GraphicalModel<double, Adder, OPENGM_TYPELIST_2(
      ExplicitFunction<double> , PottsFunction<double>), Space> Model;
   Model gm(space);

   // for each node (x, y) on the grid, i.e. for each variableIndex(x, y), add
   // one first−order functions and one first−order factor
   for(size_t y = 0; y < ny; ++y)
   for(size_t x = 0; x < nx; ++x) {
      const size_t shape[] = {numberOfLabels};
      ExplicitFunction<double> f(shape, shape + 1);
      for(size_t s = 0; s < numberOfLabels; ++s) {
         f(s) = (1.0 − lambda) * rand() / RAND_MAX;
      }
      Model::FunctionIdentifier fid = gm.addFunction(f);

      size_t variableIndices[] = {variableIndex(x, y)};
      gm.addFactor(fid, variableIndices, variableIndices + 1);
   }

   // add one (!) 2nd order Potts function
   PottsFunction<double> f(numberOfLabels, numberOfLabels, 0.0, lambda);
   Model::FunctionIdentifier fid = gm.addFunction(f);

   // for each pair of nodes (x1, y1), (x2, y2) adjacent on the grid,
   // add one factor that connecting the corresponding variable indices
```

```
    for(size_t y = 0; y < ny; ++y)
    for(size_t x = 0; x < nx; ++x) {
        if(x + 1 < nx) { // (x, y) -- (x + 1, y)
            size_t variableIndices[] = {variableIndex(x, y), variableIndex(x + 1, y)};
            sort(variableIndices, variableIndices + 2);
            gm.addFactor(fid, variableIndices, variableIndices + 2);
        }
        if(y + 1 < ny) { // (x, y) -- (x, y + 1)
            size_t variableIndices[] = {variableIndex(x, y), variableIndex(x, y + 1)};
            sort(variableIndices, variableIndices + 2);
            gm.addFactor(fid, variableIndices, variableIndices + 2);
        }
    }
}
```

2.8 Graphical Model Interface

The class template GraphicalModel inherits a rich set of member functions from the class template FactorGraph, the main interface for analyzing the structure (syntax) of the graphical model:

```
template<class S>
class FactorGraph {
public:
    typedef AccessorIterator<VariableAccessor, true> ConstVariableIterator;
    typedef AccessorIterator<FactorAccessor, true> ConstFactorIterator;

    size_t numberOfVariables() const;
    size_t numberOfVariables(const size_t) const;
    size_t numberOfFactors() const;
    size_t numberOfFactors(const size_t) const;
    size_t variableOfFactor(const size_t, const size_t) const;
    size_t factorOfVariable(const size_t, const size_t) const;

    ConstVariableIterator variablesOfFactorBegin(const size_t) const;
    ConstVariableIterator variablesOfFactorEnd(const size_t) const;
    ConstFactorIterator factorsOfVariableBegin(const size_t) const;
    ConstFactorIterator factorsOfVariableEnd(const size_t) const;
    bool variableFactorConnection(const size_t, const size_t) const;
    bool factorVariableConnection(const size_t, const size_t) const;
    bool variableVariableConnection(const size_t, const size_t) const;
    bool factorFactorConnection(const size_t, const size_t) const;
    bool isAcyclic() const;
    bool isConnected(marray::Vector<size_t>&) const;
    bool isChain(marray::Vector<size_t>&) const;
    bool isGrid(marray::Matrix<size_t>&) const;

    size_t maxFactorOrder() const;
    bool maxFactorOrder(const size_t) const;

    void variableAdjacencyMatrix(marray::Matrix<bool>&) const;
    void variableAdjacencyList(std::vector<std::set<size_t> >&) const;
    void variableAdjacencyList(std::vector<RandomAccessSet<size_t> >&) const;
    void factorAdjacencyList(std::vector<std::set<size_t> >&) const;
    void factorAdjacencyList(std::vector<RandomAccessSet<size_t> >&) const;
}
```

Arguably the most important member function of GraphicalModel is

```
template<class Iterator> ValueType evaluate(Iterator) const;
```

Its purpose is to evaluate the modeled function for a given labeling:

```
unsigned char labeling[] = {0, 1, 0};
Model::ValueType value = gm.evaluate(labeling);
```

Of course, the factors of a graphical model can also be evaluated independently, as described in Section 2.6.

2.9 Loading and Saving Graphical Models

One design goal of OpenGM was to simplify the exchange of graphical models by the introduction of a transparent file format. The result is a binary HDF5 format that is sufficiently general to accomodate the different ways of encoding discrete spaces and functions, including custom ones. It can be read by the OpenGM command line tools (Section 4), MATLAB, Python, R and many other environments that support HDF5.

Loading and saving graphical models is straight forward. Related functions are contained in a designated header file and namespace:

```
#include "opengm/graphicalmodel/graphicalmodel_hdf5.hxx"

opengm::hdf5::save(gm, "gm.h5", "toy-gm");

Model gm2;
opengm::hdf5::load(gm2, "gm.h5","toy-gm");
```

The high-level functions that are used in OpenGM to create HDF5 files as well as to load and to save multi-dimensional arrays are contained in the header file opengm/datastructures/marray/-marray_hdf5.hxx. The user is encouraged to use these for their own purposes.

Chapter 3

Inference Algorithms

OpenGM is built around the idea of separating graphical models from algorithms that operate on these. Every algorithm is a separate class template whose control parameters are summarized in a nested class called Parameter. The algorithm iterface facilitates the injection of custom code via visitors. Examples are verbose visitors that print intermediate states of algorithms to std::cout and timing visitors that measure runtime, monitor and save the progress and intermediate states of algorithms. The visitor interface is optional and comes at no runtime overhead if it is not used.

3.1 Mathematical Foundation

Given a graphical model (Section 2.1) and, instead of just the commutative monoid $(\Omega, \odot, 1)$, a commutative semi-ring $(\Omega, \odot, 1, \oplus, 0)$, the problem of computing

$$\bigoplus_{x \in X} \varphi(x) \qquad \text{i.e.} \qquad \bigoplus_{x \in X} \bigodot_{f \in F} \varphi_{\gamma(f)} \left(x_{v_1^{(f)}}, \ldots, x_{v_{|\mathcal{N}(f)|}^{(f)}} \right) \tag{3.1}$$

is a central problem in machine learning with instances in marginalization $(\mathbb{R}^+, \cdot, 1, +, 0)$, optimization $(\mathbb{R}, +, 0, \min, \infty)$, and constrained satisfaction $(\{0,1\}, \wedge, 1, \vee, 0)$:

$$\sum_{x \in X} \prod_{f \in F} \varphi_{\gamma(f)} \left(x_{v_1^{(f)}}, \ldots, x_{v_{|\mathcal{N}(f)|}^{(f)}} \right) \tag{3.2}$$

$$(\arg)\min_{x \in X} \sum_{f \in F} \varphi_{\gamma(f)} \left(x_{v_1^{(f)}}, \ldots, x_{v_{|\mathcal{N}(f)|}^{(f)}} \right) \tag{3.3}$$

$$\bigvee_{x \in X} \bigwedge_{f \in F} \varphi_{\gamma(f)} \left(x_{v_1^{(f)}}, \ldots, x_{v_{|\mathcal{N}(f)|}^{(f)}} \right) \tag{3.4}$$

For any semi-ring $(\Omega, \odot, 1, \oplus, 0)$, marginals w.r.t. a subset $U \subseteq V$ of fixed variables and corresponding labels $l_u \in X_u$ for each $u \in U$ are defined as the accumulation over the set $X' = \{x \in X | \forall u \in U : x_u = l_u\}$:

$$\bigoplus_{x \in X'} \bigodot_{f \in F} \varphi_{\gamma(f)} \left(x_{v_1^{(f)}}, \ldots, x_{v_{|\mathcal{N}(f)|}^{(f)}} \right) \ . \tag{3.5}$$

3.2 Inference Algorithm Interface

In OpenGM, all inference algorithms are derived from the class template opengm::Inference. Its two template parameters specify 1. the graphical model that determines the operation w.r.t. which the objective function factorizes and 2. the accumulative operation used to do inference (Tab. 3.1).

	opengm::Adder	opengm::Multiplier	opengm::Or
opengm::Maximizer	energy maximization	probability maximization	-
opengm::Minimizer	energy minimization	(probability minimization)	-
opengm::Adder	-	marginalization	-
opengm::And	-	-	satisfiability

Table 3.1: Useful combination of OpenGM operations.

```
namespace opengm {
    enum InferenceTermination {
        UNKNOWN=0, NORMAL=1, TIMEOUT=2, CONVERGENCE=3, INFERENCE_ERROR=4
    };
    template <class GM, class ACC>
    class Inference {
    public:
        virtual std::string name() const = 0;
        virtual const GM& graphicalModel() const = 0;
        virtual InferenceTermination infer() = 0;
        virtual void setStartingPoint(typename std::vector<LabelType>::const_iterator);
        virtual InferenceTermination arg(std::vector<LabelType>& arg, const size_t argIndex = 1) const;
        virtual InferenceTermination args(std::vector<std::vector<LabelType> >& args) const;
        virtual InferenceTermination marginal(const size_t variableID, IndependentFactorType&) const;
        virtual InferenceTermination factorMarginal(const size_t factorID, IndependentFactorType&) const;
        virtual ValueType bound() const;
        virtual ValueType value() const;
   };
}
```

The first step towards running an inference algorithm is to construct (instantiate) an object of the corresponding class, providing, as input parameters, a graphical model and, optionally, an object of the nested class Parameter. Some algorithms can be initialized via the member function setStartingPoint that takes as input an iterator to the beginning of a sequence of labels, one for each variable in the model.

Algorithms are invoked by the member function infer that returns an InferenceTermination object indicating why inference has terminated, commonly NORMAL. TIMEOUT and CONVERGENCE are specializations of NORMAL.

Results are accessed through additional member functions. Provided that the accumulative operation is associated with a linear order of the domain Ω, such as $\leq$ and $\geq$ in case of max and min, arg and value provide access to an optimal labeling and the corresponding value of the objective function. Some algorithms implement an additional member functions arg that provides access to the n best labelings and bound for a bound. If no bound is available, this function returns the neutral element of the accumulative operation. If the algorithm supports the computations of marginals (cf. (3.5) in Section 3.1), these can be accessed through the member functions marginal and factorMarginal. Marginals are provided conveniently, as independent factors (Section 2.6.1).

```
typename GraphicalModel<double, opengm::Adder> Model;
typename AnyInfernceAlgorithm<Model, opengm::Minimizer> Algorithm;

// build the model
Model gm(...);

// construct an instance of the algorithm
Algorithm::Parameter parameter;
parameter.numberOfIterations = 1000;
Algorithm algorithm(gm, parameter);

// run algorithm
algorithm.infer();

// access results
```

```
std::cout << algorithm.name() << " has found the labeling ";
std::vector<Model::LabelType> x;
algorithm.arg(x);
for(size_t j = 0; j < x.size(); ++j) {
    std::cout << x[i] << ' ';
}
std::cout << "\b. value: " << algorithm.value()
    << ". bound: " << algorithm.bound() << std::endl;
```

OpenGM provides a visitor framework to monitor the progress of inference algorithms. Visitor classes are taken as an input parameter of the function infer. They are used, for instance, by the command line tool to log upper and lower bounds over time.

Inference algorithms included in OpenGM are introduces in the following sections, together with some very basic background information and working examples which the reader is encouraged to "copy and paste".

3.3 Message Passing

A large class of inference algorithms can be formulated in a message passing framework. The main operation in this framework is to calculate and propagate messages between subsets of variables. These propagations can often be understood as a re-parameterization of the original problem with the goal of establishing special properties in the re-weighted function that makes inference easier. Even if these properties correspond to fixed points of a message passing algorithm, it need not always converge to these.

In OpenGM, the message passing framework is summarized in the class template MessagePassing. It provides multiple message passing schedules: parallel, sequential and optimized for acyclic graphs. The template parameter UpdateRules determines how messages are computed. Within this framework, OpenGM implements variants of belief propagation and tree-re-weighted belief propagation. Note that both can be used with any semi-ring, always provide marginals, and afford an estimated optimal labeling if the accumulative operation is associated with a linear order.

3.3.1 Belief Propagation

Belief propagation [31, 30, 26] is a message passing algorithm [26] that is applicable to a wide range of models. The implementation in OpenGM is based on [26], with optional message damping due to [30]. It has three control parameters, a maximum number of iterations, a convergence bound and a damping factor between zero and one, where zero means no damping. It can be used like this:

```
#include <opengm/inference/messagepassing.hxx>
#include <opengm/operations/maximizer.hxx>

// set up the optimizer (loopy belief propagation)
typedef BpUpdateRules<Model, Maximizer> UpdateRules;
typedef MessagePassing<Model, Maximizer, UpdateRules, MaxDistance> BeliefPropagation;
const size_t maxNumberOfIterations = 100;
const double convergenceBound = 1e-7;
const double damping = 0.0;
BeliefPropagation::Parameter parameter(maxNumberOfIterations, convergenceBound, damping);
BeliefPropagation bp(gm, parameter);

// optimize (approximately)
BeliefPropagation::VerboseVisitorType visitor;
bp.infer(visitor);

// obtain the (approximate) argmax
vector<size_t> labeling(numberOfVariables);
bp.arg(labeling);
```

The class Maximizer (lines 4–5) decides that the objective is to maximize the modeled function. Alternatives are Minimizer and Integrator. The class MaxDistance (line 5) is a metric for measuring the discrepancy between messages passed in consecutive iterations. If its value falls below the convergence bound, the algorithm stops. The visitor (line 14) injects code into the loop of the algorithm that prints intermediate states to std::cout. If no output is required, the visitor can be omitted in line 15 to achieve maximum performance, bp.infer();

While this implementation of Belief Propagation is very general and can deal with arbitrary factor graphs and and functions, significant speed-ups are achievable in more restrictive settings such as grid graphs. OpenGM makes the implementations of MRF-LIB and libDAI available through its interface. Yet another sequential variant of Belief Propagation is provided through the TRW-S interface.

3.3.2 TRBP

Tree Re-Weighted Belief Propagation [35] is a message passing algorithm similar to Belief Propagation. In contrast to Belief Propagation, it is based on a decomposition of the original problem into a set of acyclic sub-problems. If a re-weighting of these sub-problems exists such that the respective solutions are consistent, the original problem has been solved. However, such a re-weighting need not exists. The problem of re-weighting itself is formulated as as message passing schedule, similarly as in Belief Propagation. In addition, the expected occurrence of factors in the set of sub-problems is taken into account during the computation of messages. The set of sub-problems (subgraphs) can be passed as parameter to the algorithm or be selected automatically. In the latter case, spanning-trees are generated until all factors are covered at least once.

Marginals computed by TRBP are called pseudo-marginals and are approximations to the correct marginals. The theoretical background is described comprehensively in [35].

Our implementation of TRBP does not compute bounds on the objective. This would require additional computations for each subproblem which we have decided to avoid so far.

```
#include <opengm/inference/messagepassing.hxx>
#include <opengm/operations/maximizer.hxx>

// set up the optimizer (tree re-weighted belief propagation)
typedef TrbpUpdateRules<Model, Maximizer> UpdateRules;
typedef MessagePassing<Model, Maximizer, UpdateRules, MaxDistance> TRBP;
const size_t maxNumberOfIterations = 100;
const double convergenceBound = 1e-7;
const double damping = 0.0;
TRBP::Parameter parameter(maxNumberOfIterations, convergenceBound, damping);
TRBP trbp(gm, parameter);

// optimize (approximately)
TRBP::VerboseVisitorType visitor;
trbp.infer(visitor);

// obtain the (approximate) argmax
vector<size_t> labeling(numberOfVariables);
trbp.arg(labeling);
```

3.4 TRW-S*

TRW-S [24] is a very efficient, sequential variant of TRBP that is applicable only to second order models. It is based on a decomposition of the model into acyclic graphs all of which respect a linear order on the set of variables. If two variables are connected in a subgraph, the path between both has to be monotone w.r.t. the order. This rather technical condition typically results in sparser sub-problems and allows for a very efficient implementation of block coordinate ascent in which previous computations can be re-used. In fact, TRW-S attempts to solve the LP-relaxation of the discrete problem over the local polytope. While the dual function, which is optimized, is

non-smooth, fixed points can also coincide with points from which only combinations of block-coordinate steps would lead to improvements. These are called points of weak-tree-agreement. Thanks to its special form, TRW-S computes bounds without additional effort and includes an advanced rounding scheme to obtain integer solutions.

In OpenGM, we interface the fast implementation of TRW-S by Kolmogorov [24] that is limited to the min-sum semi-ring. In addition to the explicit functions and truncated metrics provided by the original code, we implement a view to allows TRW-S to operate directly on OpenGM data structures, without the need to duplicate the model. To use this interface, one has to download and patch Kolmogorov's original code [24] (Section 1.3).

The parameters of the TRW-S wrapper in OpenGM are: numberOfIterations_, a flag useRandomStart_ to initialize messages randomly, a flag doBPS_ to apply sequential belief propagation (BP-S) instead of TRW-S, the type of the function used by TRW-S energyType_, which can take the values VIEW, TABLES, TL1, TL2, and the tolerance for the relative duality gap tolerance_ which causes TRW-S to stop if $|E(x) - B(x)| / \max\{|E(x)|, 1\} \leq T$.

```
#include <stdlib.h>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/inference/external/trws.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>

typedef opengm::GraphicalModel<float, opengm::Adder > GraphicalModelType;
typedef opengm::external::TRWS<GraphicalModelType> TRWS;

GraphicalModelType gm;
// ...
TRWS trws(gm);
trws.infer();
std::cout << "value: " << trws.value() << " bound: " << trws.bound() << std::endl;
```

3.5 Dynamic Programming

While Belief Propagation finds an optimal labeling for acyclic models, it effectively mimics dynamic programming and performing unnecessary computations. If one is interested only in an optimal labeling and the corresponding value of the objective function and not in any marginals, an explicit implementation of dynamic programming is preferable. OpenGM provides a solution that is currently limited to second order models.

```
#include <stdlib.h>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/inference/dynamicprogramming.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>

typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::DynamicProgramming<GraphicalModelType, opengm::Minimizer> DP;

GraphicalModelType gm;
// ...
DP dp(gm);
dp.infer();
std::cout << "value: " << dp.value() << std::endl;
```

3.6 Dual Decomposition

OpenGM provides a rigorous implementation of the techniques of dual decomposition. The main idea is to duplicate variables in order to simplify the structure of the model. For each duplicate an additional constraint is required that ensures that both duplicates take the same state. This

constraints are in-cooperate via dual variables into the model, for a more technical formulation see [19].

After introducing duplicates each factor of the original graph is placed at least one time in the graph with the duplicates. If it is placed several times the energy is spitted. The new model often consists of several subgraphs which are not connected among each other and therefor can be optimized dependently. Typically, the construction is done in a way that the subproblems have a special form, e.g. they are acyclic an optimization becomes tractable. This is important, because optimization of the dual – which again can be understood as a re-weighting – requires optimization to calculate a subgradient at a given dual point.

The general parameters for all dual decomposition based methods are: The type of the decomposition decompositionId_ which can be set manual MANUAL by the parameter decomposition_, a automatic decomposition into a single tree TREE or a set of spanning trees SPANNINGTREES. The maximal order of dual variables can be controlled by the parameter maximalDualOrder_. If it is set to 1, no dual variables are introduced for shared factors with an larger order than 1. The maximal number of dual steps can be set by the parameter maximalNumberOfIterations_ and stopping criteria based on the absolute and relative gap can be set by minimalAbsAccuracy_ and minimalRelAccuracy_, respectively.

For the updates on the dual function OpenGM provides two methods: (i) projected subgradient methods and (ii) bundle methods.

3.6.1 Using Sub-Gradient Methods

The naive subgradient method start from the dual point 0 and iteratively make steps into the direction of a subgradient s. The three mayor parameters to control the scale of the scaling τ^t of the update are the stepsize-stride stepsizeStride_ (τ), -scale stepsizeScale_ (β) and -exponent stepsizeExponent_ (α), defining the stepsize

$$\tau^t = \tau \cdot \frac{1}{1 + (\beta \cdot t)^\alpha}.$$

By setting the parameter stepsizeNormalizedSubgradient_ additional normalization by the length of the subgradient is performed such that

$$\tau^t = \tau \cdot \frac{1}{1 + (\beta \cdot t)^\alpha} \frac{1}{\|s\|_2}.$$

Under some mild conditions [19] both update rules are guarantee to converge. But practically they are to slow. Komodakis [25] suggest to use an adaptive stepsize, which can be used by setting the parameter useAdaptiveStepsize_, causing

$$\tau^t = \tau \cdot \frac{|\bar{E} - B|}{\|s\|_2}.$$

Since this does not work well for use, we prefer an projected adaptive stepsize, which also makes more sense from a theoretical point of view. This can be used by setting the parameter useProjectedAdaptiveStepsize_ and cause

$$\tau^t = \tau \cdot \frac{|\bar{E} - B|}{\|s\|_2^2}.$$

Setting up a dual decomposition algorithm is a little bit more complicated, but allows to use any inference method for the subproblems. The template parameters are the graphical model type, the inference algorithm for the subproblems and the class including the dual blocks.

```
#include <stdlib.h>                                                    1
#include <opengm/graphicalmodel/graphicalmodel.hxx>                    2
#include <opengm/inference/messagepassing.hxx>                         3
```

```
#include <opengm/inference/dualdecomposition/dualdecomposition_subgradient.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>

typedef opengm::Minimizer AccType;
typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::DDDualVariableBlock<marray::Marray<ValueType> > DualBlockType;
typedef opengm::DualDecompositionBase<GraphicalModelType,DualBlockType>::SubGmType SubGmType;
typedef opengm::BeliefPropagationUpdateRules<SubGmType, AccType> UpdateRuleType;
typedef opengm::MessagePassing<SubGmType, AccType, UpdateRuleType, opengm::MaxDistance> InfType;
typedef opengm::DualDecompositionSubGradient<GraphicalModelType,InfType,DualBlockType>
DualDecompositionSubGradient;

GraphicalModelType gm;
// ...
DualDecompositionSubGradient ddsg(gm);
ddsg.infer();
std::cout << "value: " << ddsg.value() << " bound: " << ddsg.bound() << std::endl;
```

3.6.2 Using Bundle Methods*

As suggested in [19] OpenGM also provides an interface to use bundle methods for dual optimization. To use our bundle-solver, one has to install the Conic Bundle Library by Christoph Helmberg, available under the GPL2. After downloading the library one has to compile it and set the path in cmake and activate WITH_BUNDLE.

The idea of the bundle-method is to build a linear approximation of the dual function around the current working point together with a trust region term, that keeps optimization local. See [19] and the references their in for technical details. While compared to subgradient methods, bundle-methods requires to solve a small QP for updating the dual, the number of iterations required is typically much less, such that overall bundle-methods performs faster. Furthermore there seemed to be more robust with respect to the choice of parameters.

Parameters for the bundle method are: The maximal size of bundle maxBundlesize_, upper and lower bound on the trust region weight minDualWeight_ and maxDualWeight_. Furthermore, the flag activeBoundFixing_ can be used to fix variables automatically to the center value if their bounds are strongly active, noBundle_ to use a special solver that only employs a minimal bundle consisting of just one new and one aggregate gradient so that there is no real bundle available, useHeuristicStepsize_ to uses heuristic for stepsize/trustregion-radius by Kiewel.

```
#include <stdlib.h>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/inference/messagepassing.hxx>
#include <opengm/inference/dualdecomposition/dualdecomposition_bundle.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>

typedef opengm::Minimizer AccType;
typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::DDDualVariableBlock<marray::Marray<ValueType> > DualBlockType;
typedef opengm::DualDecompositionBase<GraphicalModelType,DualBlockType>::SubGmType SubGmType;
typedef opengm::BeliefPropagationUpdateRules<SubGmType, AccType> UpdateRuleType;
typedef opengm::MessagePassing<SubGmType, AccType, UpdateRuleType, opengm::MaxDistance> InfType;
typedef opengm::DualDecompositionBundle<GraphicalModelType,InfType,DualBlockType>
DualDecompositionBundle;

GraphicalModelType gm;
// ...
DualDecompositionBundle ddb(gm);
ddb.infer();
std::cout << "value: " << ddb.value() << " bound: " << ddb.bound() << std::endl;
```

3.7 A^* Search

The A^* search algorithm suggested in [4, 18] is a branch-and-bound algorithm. For the calculation of the bound a tree-based approximation is used which guarantee to bound the optimal objective of a subset of states. For this highly efficient calculation of bounds the set of allowed branches is restricted and predefined by the parameters of the algorithm. While the method does not scale to large scale problems, it is state-of-the-art for models with a moderate size of variables (up to 100). The density of the model and the number of labels are from minor important. Especially when the pairwise and higher order terms are highly discriminative, the A^*-method performs fast. In general it guarantee to find the best or N-best states, but has no guarantee for polynomial runtime.

It is very important to select the node order and the tree used for the bound calculation appropriate. The version 2.0 includes no reasonable automatic selection – we hope to include this in a later release – especially if your models have a irregular topology one can gain a speed up of several magnitudes. For large models it would make sense to restrict the maximal heap size. If the maximal heap size is reached, the worst subsets are dropped. Since their bound is known, it may be still possible to guarantee optimality, see [4, 18] for details. All parameters listed below:

maxHeapSize_	maximum size of the heap
numberOfOpt_	number of N-best solutions that should be found
objectiveBound_	states that a worse are not considered as solution
heuristic_	method used to calculate bound/heuristic
→ DEFAULTHEURISTIC	... select fastest one automatically
→ FASTHEURISTIC	... use dynamic programming for second order models
→ STANDARDHEURISTIC	... use belief propagation
nodeOrder_	vector containing the nodes in the order they should be proceed
treeFactorIds_	factor ids spanning the tree used for bound calculation

```
#include <stdlib.h>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/inference/astar.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>

typedef opengm::Minimizer AccType;
typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::AStar<GraphicalModelType, opengm::Minimizer> ASTAR;

GraphicalModelType gm;
// ...
ASTAR astar(gm);
astar.infer();
std::cout << "value: " << astar.value() << " bound: " << astar.bound() << std::endl;
```

3.8 (Integer) Linear Programming*

Any energy minimization problem can be formulated as an (integer) linear program. Often it is also sufficient to consider a relaxation of the original problem by relaxing the system of inequalities. The common supposition that such problem formulations are generally computationally intractable is not true! OpenGM provides a wrapper that transform such problems in either an LP-relaxation with respect to the first order local polytope [35] or an integer linear programming formulation which is solved via branch-and-cut.

To use these methods, CPLEX [15] needs to be installed. Versions free for academic use can be obtained through the IBM academic initiative. Once IBM ILOG CPLEX and IBM CONCERT are installed, the respective paths need to be set in the OpenGM CMake environment and activated by checking WITH_CPLEX. After a re-build, LP and ILP formulations and the CPLEX solver are available via the class opengm::LPCplex.

The corresponding parameter class takes many optional parameters which can be used to tune the solver. The most important for beginners are the flag to switch to integer mode integerConstraint_, the number of threads that should be used numberOfThreads_ (0=autosect), the verbose flag verbose_ that switches the CPLEX output on, and the time limit timeLimit_ that stops the method after N seconds realtime.

```
#include <stdlib.h>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>
#include <opengm/inference/lpcplex.hxx>

typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef LPCplex<GraphicalModelType, opengm::Minimizer> LPCPLEX;
GraphicalModelType gm;
// ...

//Set up LP
LPCPLEX lp(gm);
lp.infer();
std::cout << "value: " << lp.value() << " bound: " << lp.bound() << std::endl;

//Set up ILP
typename LPCPLEX::Parameter para;
para.integerConstraint_ = true;
para.timeLimit_ = 60.0;
para.verbose_ = true;
para.numberOfThreads_ = 4;
LPCPLEX ilp(gm,para);
ilp.infer();
std::cout << "value: " << ilp.value() << " bound: " << ilp.bound() << std::endl;
```

3.9 Graph Cuts*

OpenGM comes with a framework for optimization by graph cuts [8], for graphical models whose operation is addition, whose variables are binary[1] and whose factors are submodular and of order at most 3. It can be used with the different st-min-cut/max-flow algorithms from the boost graph library, by Kolmogorov [21] and an implementation of *Maximum Flows By Incremental Breadth First Search* (IBFS) [10] by Hed [13]. As an example, consider a graphical model gm that has the necessary properties, and the following graph cut optimizer:

```
#include <iostream>

#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/inference/graphcut.hxx>
#include <opengm/inference/auxiliary/minstcutkolmogorov.hxx>
#include <opengm/operations/minimizer.hxx>

typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::external::MinSTCutKolmogorov<size_t, double> MinCut;
typedef opengm::GraphCut<GraphicalModelType, opengm::Minimizer, MinStCutType> MinCut;

MinCut mincut(gm);
mincut.infer();
std::cout << "value: " << mincut.value() << std::endl;
```

The next example uses a push-relabel algorithm from the boost graph library that is designed for integer labeled graphs[2]. The fractional values of the graphical model are scaled by 1000000 and truncated.

[1] For submodular functions on variables with more then two labels, consider the move making algorithms described in Section 3.11.

[2] IBFS is also designed for integer-valued max-flow problems.

```
#include <iostream>

#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/inference/graphcut.hxx>
#include <opengm/inference/auxiliary/minstcutboost.hxx>
#include <opengm/operations/minimizer.hxx>

typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::MinSTCutBoost<size_t, long, opengm::PUSH_RELABEL> MinStCutType;
typedef opengm::GraphCut<GraphicalModelType, opengm::Minimizer, MinStCutType> MinGraphCut;

MinGraphCut::Parameter para(1000000);
MinCut mincut(gm,para);
mincut.infer();
std::cout << "value: " << mincut.value() << std::endl;
```

3.10 QPBO*

QPBO (quadratic pseudo-boolean optimization) is based on the principle of roof duality [11] and was proposed for unconstraint quadratic binary programming and network-flow problems by Boros and Hammer [6, 7]. Its first application to graphical models is due to Rother et al. [32]. OpenGM wraps the original code by Kolmogorov [22] and, in addition, provides its own QPBO solver that builds the corresponding max-flow problem, solves it with a graph cut solver and transform the resulting flow and cut back into a solution of the original problem. We suggest to use Kolmogorov's code which is faster.

QPBO can be applied on any second order binary model. If the original problem is permuted submodular, it is solved to optimality. In general, QPBO yields the solution of the LP relaxation over the local polytope which is half-integral, i.e. each attains a label 0, 1 or $\frac{1}{2}$. Furthermore, QPBO provides a partial optimal labeling for some variables, a property known as persistency [32]. Even if the complete optimal labeling remains unknown, parts of the problem can often be solved to optimality.

```
#include <iostream>

#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/inference/external/qpbo.hxx>
#include <opengm/operations/minimizer.hxx>

typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::external::QPBO<GraphicalModelType> QPBO;

GraphicalModelType gm;
// ...
QPBO qpbo(gm);
qpbo.infer();

std::vector<bool> optimalVariables;
double partialOpt = partialOptimality(optimalVariables);
std::cout << "value: " << qpbo.value() << " bound: " << qpbo.bound() << std::endl;
std::cout << "partial optimality: " << partialOpt << std::endl;
```

3.11 Move Making

OpenGM provides several algorithms that attempt to improve an initial labeling by moving to better labelings via a constrained set of moves. This class of algorithms include α-Expansion [8] (AlphaExpasion), $\alpha\beta$-Swap [8] (AlphaBetaSwap), ICM [5] (ICM), and the Lazy Flipper [1] (LazyFlipper).

3.11.1 α-Expansion

The α-expansion algorithm [8] starts from an initial labeling and continues to iterates over all labels α that variables can attain. In each iteration, a problem with binary variables is constructed from the original problem, based on the following question: Which subset of variables whose current label is not α should be labeled α to yield an optimal improvement w.r.t. the current labeling? In OpenGM, any algorithm can be used for the binary sub-problems. It should be chosen such that sub-problems are solved to optimality. Under mild conditions [8], each move can be reduced to a max-flow problem and solved efficiently. If no improvements are possible for any α, the algorithm has converged.

```
#include <iostream>

#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>
#include <opengm/inference/graphcut.hxx>
#include <opengm/inference/alphaexpansion.hxx>
#include <opengm/inference/auxiliary/minstcutkolmogorov.hxx>

typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::external::MinSTCutKolmogorov<size_t, double> MinStCutType;
typedef opengm::GraphCut<GraphicalModelType, opengm::Minimizer, MinStCutType> MinGraphCut;
typedef opengm::AlphaExpansion<GraphicalModelType, MinGraphCut> MinAlphaExpansion;

GraphicalModelType gm;
// ...
MinAlphaExpansion ae(gm);
ae.infer();

std::cout << "value: " << ae.value() << std::endl;
```

More details on the parameters can be found in the reference documentation. For grid graphs, OpenGM provides an interface to the MRF-LIB (Section 3.14.2) which contains the original implementation of [8] that is more efficient for these types of problems.

3.11.2 $\alpha\beta$-Swap

The $\alpha\beta$-swap algorithm [8] is similar to α-expansion, except that it iterated over all pairs of distinct labels α and β and, in each iteration, a binary problem is constructed based on the question which variables that are currently labeled α should be labeled β such that the improvement over the current labeling is optimal. All other variables are fixed. If no $\alpha\beta$-swap can improve the current labeling, the algorithm has converged. Under mild conditions [8], each move can be reduced to a max-flow problem and solved efficiently. The implementation in OpenGM allows any algorithm to be used for the sub-problems. It should solve the sub-problems to optimality.

```
#include <iostream>

#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>
#include <opengm/inference/graphcut.hxx>
#include <opengm/inference/alphabetaswap.hxx>
#include <opengm/inference/auxiliary/minstcutkolmogorov.hxx>

typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::external::MinSTCutKolmogorov<size_t, double> MinStCutType;
typedef opengm::GraphCut<GraphicalModelType, opengm::Minimizer, MinStCutType> MinGraphCut;
typedef opengm::AlphaBetaSwap<GraphicalModelType, MinGraphCut> MinAlphaBetaSwap;

GraphicalModelType gm;
// ...
MinAlphaBetaSwap abs(gm);
abs.infer();

std::cout << "value: " << abs.value() << std::endl;
```

Further details on the parameters can be found in the reference documentation. For grid graphs, OpenGM provides an interface to the MRF-LIB (Section 3.14.2) that contains the original implementation of [8] and is more efficient for these types of problems.

3.11.3 Iterative Conditional Modes (ICM)

In ICM [5], one iterates over the variables. For each variable, the optimal labeling is chosen, conditioned to all other variables being fixed. The process is repeated until none change of a single label leads to an improvement[3]. This method is fast and simple. However, its fixed points are only guaranteed to be optimal within a Hamming distance of 1 which makes the algorithm prone to local optima and sensitive to the initialization.

```
#include <iostream>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>
#include <opengm/inference/icm.hxx>

typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::ICM<GraphicalModelType, opengm::Minimizer> ICM;
typedef GraphicalModelType::LabelType LabelType;

GraphicalModelType gm;
// ...
std::vector<LabelType> startPoint;
//..
ICM::Parameter para(startPoint);
ICM icm(gm,para);
icm.infer();

std::cout << "value: " << icm.value() << std::endl;
```

3.11.4 Lazy Flipper

Similar to ICM, the Lazy Flipper tries to improve a initial labeling sequentially. Instead of considering only a single variable at a time, it takes into account also large connected subsets of variables up to a prescribed size. For a subgraph size of one, it reduce to ICM. When the Lazy Flipper converges, the labeling is guaranteed to be optimal within a Hamming distance equal to the subgraph size, i.e. no better labeling exists that can be reached from the current labeling by flipping the same number of variables or less. The algorithm keeps track of the history of visited subgraphs and successful flips.

```
#include <iostream>

#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>
#include <opengm/inference/lazyflipper.hxx>

typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::LazyFlipper<GraphicalModel, opengm::Minimizer> LazyFlipper;
typedef GraphicalModelType::LabelType LabelType;

GraphicalModelType gm;
// ...
size_t maxSubgraphSize = 2;
std::vector<LabelType> startPoint;
//..
LazyFlipper::Parameter para(maxSubgraphSize,startPoint);
LazyFlipper lf(gm,para);
```

[3] The implementation in OpenGM is more efficient.

```
lf.infer();
std::cout << "value: " << lf.value() << std::endl;
```

3.11.5 LOC

LOC [17] is a stochastic move making algorithm that starts from an arbitrary initial labeling. In each iteration, a connected subset of variables is selected at random. The problem is then solved exactly for this subset, keeping all other variables fixed. The parameters of the algorithm are phi, the parameter of the truncated geometric distribution form which a subgraph radius is drawn, maxRadius, the maximum such radius, maxIteration, the maximum number of iterations and aStarThreshold, the subgraph size beyond which Astar, instead of Bruteforce, is used to solve the sub-problem.

```
#include <opengm/inference/loc.hxx>
#include <opengm/operations/minimizer.hxx>

// set up the optimizer (bp)
typedef LOC<Model, Minimizer> LOC;
typedef Model::LabelType LabelType;

double phi = 0.5;
size_t maxRadius = 10;
size_t maxIterations = 100;
size_t aStarThreshold = 5;
LOC::Parameter parameter(phi,maxRadius,maxIterations,aStarThreshold);
LOC loc(gm, parameter);

// set starting point
std::vector<LabelType> startingPoint;
// assuming startingPoint has been filled
// with meaningful labels
loc.setStartingPoint(startingPoint.begin());

// optimize (approximately)
loc.infer( );

// obtain the (approximate) argmin
vector<LabelType> labeling(numberOfVariables);
bp.arg(labeling);
```

3.12 Sampling

OpenGM comes with two Markov Chain Monte Carlo algorithms, Gibbs sampling [9] and a generalization of Swendsen-Wang sampling [3]. Both methods are special cases of the Metropolis-Hastings algorithm [27] and can be used to approximate the joint probability mass function defined by a graphical model, to search for its modes as well as to approximate marginals of arbitrary subsets of variables. Internally, the algorithms generate correlated samples as states of a Markov chain. Each sample is a labeling of all variables. This Markov chain is defined by a proposal probability $q(x'|x_t)$ of moving from the current (e.g. an initial) labeling x_t to a new labeling x', or else sample the current labeling again.

3.12.1 Gibbs

In Gibbs sampling, the proposal probability is chosen such that at most one variable can change its label. This variable can either be drawn randomly from the uniform distribution over all variables or selected cyclicly. The new label is always drawn randomly from the uniform distribution over all labels the proposed variable can attain.

In each iteration, the proposed labeling x' is accepted depending on the ratio $r = p(x')/p(x_t)$ where $p(x)$ is the full probability mass function defined by the graphical model. If $r \geq 1$, the new

labeling is accepted, i.e. $x_{t+1} = x'$. Otherwise, x' is accepted with probability r. If x' is rejected, then $x_{t+1} = x_t$.

The class Gibbs can be used with graphical models in which the operation is multiplication or addition. In the case of multiplication, the class samples from the probability mass function that is proportional to the modeled function. This means that one does not need to normalize when setting up the graphical model. If the operation of the graphical model is addition and the modeled function is $f(x)$, the probability mass function is taken to be proportional to $\exp(-f(x))$.

The following example demonstrates how to search for the mode of a distribution and how to sample marginals of specific subsets of variables using the MarginalVisitor.

```
#include <vector>
#include <iostream>

#include <opengm/graphicalmodel/space/simplediscretespace.hxx>
#include <opengm/functions/explicit_function.hxx>
#include <opengm/functions/potts.hxx>
#include <opengm/operations/multiplier.hxx>
#include <opengm/operations/maximizer.hxx>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/inference/gibbs.hxx>

typedef opengm::SimpleDiscreteSpace<> Space;
typedef opengm::ExplicitFunction<double> ExplicitFunction;
typedef opengm::PottsFunction<double> PottsFunction;
typedef opengm::meta::TypeListGenerator<ExplicitFunction, PottsFunction>::type FunctionTypes;
typedef opengm::GraphicalModel<double, opengm::Multiplier, FunctionTypes, Space> GraphicalModel;
typedef opengm::Gibbs<GraphicalModel, opengm::Maximizer> Gibbs;
typedef opengm::GibbsMarginalVisitor<Gibbs> MarginalVisitor;

// build a model with 10 binary variables in which
// - the first variable is more likely to be labeled 1 than 0
// - neighboring variables are more likely to have similar labels than dissimilar
void buildGraphicalModel(GraphicalModel& gm) {
   const size_t numberOfVariables = 10;
   const size_t numberOfLabels = 2;
   Space space(numberOfVariables, numberOfLabels);
   gm = GraphicalModel(space);

   // add 1st order function
   GraphicalModel::FunctionIdentifier fid1;
   {
      ExplicitFunction f(&numberOfLabels, &numberOfLabels + 1);
      f(0) = 0.2;
      f(1) = 0.8;
      fid1 = gm.addFunction(f);
   }

   // add 2nd order function
   GraphicalModel::FunctionIdentifier fid2;
   {
      const double probEqual = 0.7;
      const double probUnequal = 0.3;
      PottsFunction f(2, 2, probEqual, probUnequal);
      fid2 = gm.addFunction(f);
   }

   // add 1st order factor (at first variable)
   {
      size_t variableIndices[] = {0};
      gm.addFactor(fid1, variableIndices, variableIndices + 1);
   }

   // add 2nd order factors
   for(size_t j = 0; j < numberOfVariables - 1; ++j) {
      size_t variableIndices[] = {j, j+1};
```

```
        gm.addFactor(fid2, variableIndices, variableIndices + 2);
    }
}

// use Gibbs sampling for the purpose of finding the most probable labeling
void gibbsSamplingForOptimization(const GraphicalModel& gm) {
    const size_t numberOfSamplingSteps = 1e4;
    const size_t numberOfBurnInSteps = 1e4;
    Gibbs::Parameter parameter(numberOfSamplingSteps, numberOfBurnInSteps);
    Gibbs gibbs(gm, parameter);
    gibbs.infer();
    std::vector<size_t> argmax;
    gibbs.arg(argmax);

    std::cout << "most probable labeling sampled: (";
    for(size_t j = 0; j < argmax.size(); ++j) {
        std::cout << argmax[j] << ", ";
    }
    std::cout << "\b\b)" << std::endl;
}

// use Gibbs sampling to estimate marginals
void gibbsSamplingForMarginalEstimation(const GraphicalModel& gm) {
    MarginalVisitor visitor(gm);

    // extend the visitor to sample first order marginals
    for(size_t j = 0; j < gm.numberOfVariables(); ++j) {
        visitor.addMarginal(j);
    }

    // extend the visitor to sample certain second order marginals
    for(size_t j = 0; j < gm.numberOfVariables() - 1; ++j) {
        size_t variableIndices[] = {j, j + 1};
        visitor.addMarginal(variableIndices, variableIndices + 2);
    }

    // sample
    Gibbs gibbs(gm);
    gibbs.infer(visitor);

    // output sampled first order marginals
    std::cout << "sampled first order marginals:" << std::endl;
    for(size_t j = 0; j < gm.numberOfVariables(); ++j) {
        std::cout << "x" << j << ": ";
        for(size_t k = 0; k < 2; ++k) {
            const double p = static_cast<double>(visitor.marginal(j)(k)) / visitor.numberOfSamples();
            std::cout << p << ' ';
        }
        std::cout << std::endl;
    }

    // output sampled second order marginals
    std::cout << "sampled second order marginals:" << std::endl;
    for(size_t j = gm.numberOfVariables(); j < visitor.numberOfMarginals(); ++j) {
        std::cout << "x" << visitor.marginal(j).variableIndex(0)
            << ", x" << visitor.marginal(j).variableIndex(1)
            << ": ";
        for(size_t x = 0; x < 2; ++x)
            for(size_t y = 0; y < 2; ++y) {
                const double p = static_cast<double>(visitor.marginal(j)(x, y)) / visitor.numberOfSamples();
                std::cout << p << ' ';
            }
            std::cout << std::endl;
    }
}

int main() {
```

```
   GraphicalModel gm;
   buildGraphicalModel(gm);
   gibbsSamplingForOptimization(gm);
   gibbsSamplingForMarginalEstimation(gm);
   return 0;
}
```

3.12.2 Swendsen-Wang

Swendsen-Wang sampling and its generalizations [3] are instances of the Metropolis-Hastings algorithm, with a proposal distribution that allows the labels of multiple variables to change in a single step. Usage of the class template SwendsenWang is similar to that of Gibbs and is shown in a separate example below.

The implementation in OpenGM is based on [3]. It considers the adjacency graph of variables in which two variables are connected if and only if (i) both variables can attain the same number of labels and (ii) there exists at least one factor that depends on at least both variables. For each such pair of variables, we estimate the joint marginal by accumulating all factors that depend on at least one of the variables, a procedure that is not practical for all graphical models. From this estimate of the marginal, we compute the probability of the two variables taking on equal and unequal labels, respectively.

In each iteration, we consider the adjacency graph of variables above and remove all edges between variables that are currently assigned different labels. From the remaining connected components, edges are removed at random, according to the probabilities estimated before. Finally, one of the now possibly even smaller connected components and a proposed new label are drawn uniformly at random, and this relabeling is subject to the Metropolis-Hastings acceptance procedure. The C++ implementation is more efficient and does not involve dynamic manipulations of graphs.

```
#include <vector>
#include <iostream>

#include <opengm/graphicalmodel/space/simplediscretespace.hxx>
#include <opengm/functions/explicit_function.hxx>
#include <opengm/functions/potts.hxx>
#include <opengm/operations/multiplier.hxx>
#include <opengm/operations/maximizer.hxx>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/inference/swendsenwang.hxx>

typedef opengm::SimpleDiscreteSpace<> Space;
typedef opengm::ExplicitFunction<double> ExplicitFunction;
typedef opengm::PottsFunction<double> PottsFunction;
typedef opengm::meta::TypeListGenerator<ExplicitFunction, PottsFunction>::type FunctionTypes;
typedef opengm::GraphicalModel<double, opengm::Multiplier, FunctionTypes, Space> GraphicalModel;
typedef opengm::SwendsenWang<GraphicalModel, opengm::Maximizer> SwendsenWang;
typedef opengm::SwendsenWangMarginalVisitor<SwendsenWang> MarginalVisitor;

// build a Markov Chain with 10 binary variables in which
// - the first variable is more likely to be labeled 1 than 0
// - neighboring variables are more likely to have similar labels than dissimilar
void buildGraphicalModel(GraphicalModel& gm) {
   const size_t numberOfVariables = 10;
   const size_t numberOfLabels = 2;
   Space space(numberOfVariables, numberOfLabels);
   gm = GraphicalModel(space);

   // add 1st order function
   GraphicalModel::FunctionIdentifier fid1;
   {
```

```
        ExplicitFunction f(&numberOfLabels, &numberOfLabels + 1);
        f(0) = 0.2;
        f(1) = 0.8;
        fid1 = gm.addFunction(f);
    }

    // add 2nd order function
    GraphicalModel::FunctionIdentifier fid2;
    {
        const double probEqual = 0.7;
        const double probUnequal = 0.3;
        PottsFunction f(2, 2, probEqual, probUnequal);
        fid2 = gm.addFunction(f);
    }

    // add 1st order factor (at first variable)
    {
        size_t variableIndices[] = {0};
        gm.addFactor(fid1, variableIndices, variableIndices + 1);
    }

    // add 2nd order factors
    for(size_t j = 0; j < numberOfVariables - 1; ++j) {
        size_t variableIndices[] = {j, j+1};
        gm.addFactor(fid2, variableIndices, variableIndices + 2);
    }
}

// use SwendsenWang sampling for the purpose of finding the most probable labeling
void swendsenWangSamplingForOptimization(const GraphicalModel& gm) {
    const size_t numberOfSamplingSteps = 1e4;
    const size_t numberOfBurnInSteps = 1e4;
    SwendsenWang::Parameter parameter(numberOfSamplingSteps, numberOfBurnInSteps);
    SwendsenWang swendsenWang(gm, parameter);
    swendsenWang.infer();
    std::vector<size_t> argmax;
    swendsenWang.arg(argmax);

    std::cout << "most probable labeling sampled: (";
    for(size_t j = 0; j < argmax.size(); ++j) {
        std::cout << argmax[j] << ", ";
    }
    std::cout << "\b\b)" << std::endl;
}

// use SwendsenWang sampling to estimate marginals
void swendsenWangSamplingForMarginalEstimation(const GraphicalModel& gm) {
    MarginalVisitor visitor(gm);

    // extend the visitor to sample first order marginals
    for(size_t j = 0; j < gm.numberOfVariables(); ++j) {
        visitor.addMarginal(j);
    }

    // extend the visitor to sample certain second order marginals
    for(size_t j = 0; j < gm.numberOfVariables() - 1; ++j) {
        size_t variableIndices[] = {j, j + 1};
        visitor.addMarginal(variableIndices, variableIndices + 2);
    }

    // sample
    SwendsenWang swendsenWang(gm);
    swendsenWang.infer(visitor);

    // output sampled first order marginals
    std::cout << "sampled first order marginals:" << std::endl;
    for(size_t j = 0; j < gm.numberOfVariables(); ++j) {
```

```
        std::cout << "x" << j << ": ";
        for(size_t k = 0; k < 2; ++k) {
            const double p = static_cast<double>(visitor.marginal(j)(k)) / visitor.numberOfSamples();
            std::cout << p << ' ';
        }
        std::cout << std::endl;
    }

    // output sampled second order marginals
    std::cout << "sampled_second_order_marginals:" << std::endl;
    for(size_t j = gm.numberOfVariables(); j < visitor.numberOfMarginals(); ++j) {
        std::cout << "x" << visitor.marginal(j).variableIndex(0)
            << ",x" << visitor.marginal(j).variableIndex(1)
            << ": ";
        for(size_t x = 0; x < 2; ++x)
            for(size_t y = 0; y < 2; ++y) {
                const double p = static_cast<double>(visitor.marginal(j)(x, y)) / visitor.numberOfSamples();
                std::cout << p << ' ';
            }
            std::cout << std::endl;
    }
}

int main() {
    GraphicalModel gm;

    buildGraphicalModel(gm);
    swendsenWangSamplingForOptimization(gm);
    swendsenWangSamplingForMarginalEstimation(gm);

    return 0;
}
```

3.13 Brute Force Search

OpenGM provides an implementation of brute force exhaustive search, mainly for the purpose of testing other algorithms and to solve small problems with minimal overhead. In fact, the header file include/opengm/inference/bruteforce.hxx is an excellent starting point for implementing custom algorithms (Section 6.4).

```
#include <stdlib.h>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>
#include <opengm/inference/bruteforce.hxx>

typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::Bruteforce<GraphicalModelType,opengm::Minimizer> Bruteforce;\\[-1.5ex]GraphicalModelType
gm;// ...Bruteforce bf(gm);bf.infer();

std::cout << "value: " << bf.value() << std::endl;
```

3.14 Wrappers around Other Libraries

3.14.1 libDAI*

OpenGM wrappers (Tab. 3.2) around the inference algorithms of libDAI [28, 29] are collected in the namespace opengm::external::libdai: These classes derive from opengm::external::libdai::Inference which itself derives from opengm::Inference. Therefore, wrapped algorithms can be used exactly like native OpenGM algorithms:

Algorithm	Wrapper class
Belief Propagation	Bp
TRBP	TreeReweightedBp
Double Loop Generalized BP	DoubleLoopGeneralizedBP
Fractional BP	FractionalBp
Loop Corrected BP	LoopCorrectedBp
Tree Expectation Propagation	TreeExpectationPropagation
Exact Inference	Exact
Junction Tree	JunctionTree
Gibbs Sampling	Gibbs
Mean Field	MeanField

Table 3.2: OpenGM classes wrapping libDAI inference algorithms.

3.14.1.1 Belief Propagation

```
#include <opengm/inference/external/libdai/bp.hxx>
#include <opengm/operations/maximizer.hxx>

// set up the optimizer (bp)
typedef external::libdai::Bp<Model, Maximizer> Bp;

const size_t maxIterations=100;
const double damping=0.0;
const double tolerance=0.000001;
// Bp::UpdateRule = PARALL | SEQFIX | SEQRND | SEQMAX
Bp::UpdateRule updateRule= PARALL;
const size_t verbose=0;
size_t verboseLevel=0;
Bp::Parameter parameter(maxIterations, damping,tolerance,updateRule,verboseLevel);
Bp bp(gm, parameter);

// optimize (approximately)
bp.infer( );

// obtain the (approximate) argmax
vector<size_t> labeling(numberOfVariables);
bp.arg(labeling);
```

3.14.1.2 TRBP

```
#include <opengm/inference/external/libdai/tree_reweighted_bp.hxx.hxx>
#include <opengm/operations/maximizer.hxx>

// set up the optimizer (trbp)
typedef external::libdai::TreeReweightedBp<Model, Maximizer> Trbp;

const size_t maxIterations=100;
const double damping=0.0;
const double tolerance=0.000001;
const size_t ntrees=10;
// Trbp::UpdateRule = PARALL | SEQFIX | SEQRND | SEQMAX
Trbp::UpdateRule updateRule= PARALL;
const size_t verboseLevel=0;
size_t verboseLevel=0;
Trbp::Parameter parameter(maxIterations, damping,tolerance,ntrees,updateRule,verboseLevel);
Trbp trbp(gm, parameter);

// optimize (approximately)
trbp.infer( );

// obtain the (approximate) argmax
vector<size_t> labeling(numberOfVariables);
trbp.arg(labeling);
```

3.14.1.3 Double Loop Generalized BP

```
#include <opengm/inference/external/libdai/fractional_bp.hxx>
#include <opengm/operations/minimzer.hxx>

// set up the optimizer (fractional bp)
typedef external::libdai::DoubleLoopGeneralizedBP<Model, Minimzer> DoubleLoopGeneralizedBP;

const bool doubleloop=1;
// DoubleLoopGeneralizedBP::Clusters=MIN | BETHE | DELTA | LOOP
const DoubleLoopGeneralizedBP::Clusters clusters=BETHE;
const size_t loopdepth = 3;
// DoubleLoopGeneralizedBP::Init = UNIFORM | RANDOM
const DoubleLoopGeneralizedBP::Init init=UNIFORM;
const size_t maxiter=10000;
const double tolerance=1e-9;
const size_t verboseLevel=0;
DoubleLoopGeneralizedBP::Parameter parameter( loopdepth, init, maxiter, tolerance,verboseLevel);
DoubleLoopGeneralizedBP gdlbp(gm, parameter);

// optimize (approximately)
gdlbp.infer( );

// obtain the (approximate) argmin
vector<size_t> labeling(numberOfVariables);
gdlbp.arg(labeling);
```

3.14.1.4 Fractional BP

```
#include <opengm/inference/external/libdai/fractional_bp.hxx>
#include <opengm/operations/maximizer.hxx>

// set up the optimizer (fractional bp)
typedef external::libdai::FractionalBp<Model, Maximizer> FractionalBp;

const size_t maxIterations=100;
const double damping=0.0;
const double tolerance=0.000001;
// FractionalBp::UpdateRule = PARALL | SEQFIX | SEQRND | SEQMAX
FractionalBp::UpdateRule updateRule= PARALL;
const size_t verboseLevel=0;
size_t verboseLevel=0;
FractionalBp::Parameter parameter(maxIterations, damping,tolerance,updateRule,verboseLevel);
FractionalBp fbp(gm, parameter);

// optimize (approximately)
fbp.infer( );

// obtain the (approximate) argmax
vector<size_t> labeling(numberOfVariables);
fbp.arg(labeling);
```

3.14.1.5 TreeExpectationPropagation

```
#include <opengm/inference/external/libdai/fractional_bp.hxx>
#include <opengm/operations/maximizer.hxx>

// set up the optimizer (fractional bp)
typedef external::libdai::TreeExpectationPropagation<Model, Maximizer> TreeExpectationPropagation;

//TreeExpectationPropagation::TreeEpType = ORG | ALT
TreeExpectationPropagation::TreeEpType treeEpTyp=ORG;
const size_t maxiter=10000,
const double tolerance=1e-9,
size_t verboseLevel=0
TreeExpectationPropagation::Parameter parameter(treeEpTyp,maxiter,tolerance,verboseLevel);
TreeExpectationPropagation treeep(gm, parameter);
```

```
// optimize (approximately)
treeep.infer( );

// obtain the (approximate) argmax
vector<size_t> labeling(numberOfVariables);
treeep.arg(labeling);
```

3.14.1.6 Exact

```
#include <opengm/inference/external/libdai/exact.hxx>
#include <opengm/operations/minimizer.hxx>

// set up the optimizer (junction tree)
typedef external::libdai::Exact<Model, Minimizer> Exact;

size_t verboseLevel=0;
Exact::Parameter parameter(verboseLevel);
Exact exact(gm, parameter);

// optimize (to global optimum)
exact.infer( );

// obtain the argmin
vector<size_t> labeling(numberOfVariables);
exact.arg(labeling);
```

3.14.1.7 Junction Tree

```
#include <opengm/inference/external/libdai/junction_tree.hxx>
#include <opengm/operations/minimizer.hxx>

// set up the optimizer (exact)
typedef external::libdai::JunctionTree<Model, Minimizer> JunctionTree;

// JunctionTree::UpdateRule = HUGIN | SHSH
JunctionTree::UpdateRule updateRule=HUGIN;
// JunctionTree::Heuristic = MINFILL | WEIGHTEDMINFILL | MINNEIGHBORS
JunctionTree::Heuristic heuristic=MINFILL;
size_t verboseLevel=0;
JunctionTree::Parameter parameter(updateRule, heuristic,verboseLevel);
JunctionTree jt(gm, parameter);

// optimize (to global optimum)
jt.infer( );

// obtain the argmin
vector<size_t> labeling(numberOfVariables);
jt.arg(labeling);
```

3.14.1.8 Gibbs

```
#include <opengm/inference/external/libdai/gibbs.hxx>
#include <opengm/operations/minimizer.hxx>

// set up the optimizer (gibbs sampler)
typedef external::libdai::Gibbs<Model, Minimizer> Gibbs;

const size_t maxiter=10000;
const size_t burnin=100;
const size_t restart_=10000;
const size_t verbose=0;
Gibbs::Parameter parameter(maxiter,burnin,restart_,verboseLevel);
Gibbs gibbs(gm, parameter);

// optimize
gibbs.infer( );

// obtain the argmin
```

```
vector<size_t> labeling(numberOfVariables);
gibbs.arg(labeling);
```

3.14.1.9 Mean Field

```
#include <opengm/inference/external/libdai/mean_field.hxx>
#include <opengm/operations/minimizer.hxx>

// set up the optimizer (mean field)
typedef external::libdai::MeanField<Model, Minimizer> MeanField;

const size_t maxiter=10000;
const double damping=0.2;
const double tolerance=1e-9;
// MeanField::UpdateRule = NAIVE | HARDSPIN
const MeanField::UpdateRule updateRule= NAIVE;
// MeanField::Init = UNIFORM | RANDOM
const MeanField::Init init=UNIFORM;
const size_t verboseLevel=0;
MeanField::Parameter parameter(maxiter,damping,tolerance,updateRule,verboseLevel);
MeanField mf(gm, parameter);

// optimize (approximately)
mf.infer( );

// obtain the (approximate) argmin
vector<size_t> labeling(numberOfVariables);
mf.arg(labeling);
```

3.14.2 MRF-LIB*

MRF-LIB [34, 33] is a highly optimized inference library. It is restricted to energy minimization problem and second-order grid models with a four neighborhood. Many important computer vision problems belong to this class. MRF-LIB implements Loopy Belief Propagation (LBP), Sequential Belief Propagation (BP-S), Tree Re-weighted Belief Propagation (TRW-S), Iterated Conditional Modes (ICM), α-Expansion, and $\alpha\beta$-Swap.

OpenGM makes all these implementations available in two alternative ways: On the one hand, a graphical model constructed in OpenGM copied into the native MRF-LIB data-structure for which highly optimized code exists. On the other hand, a view function can be used internally that makes OpenGM models look to the MRF-LIB code as if they were stored in the native data structure. No data is copied in this case. In fact, MRF-LIB can even beneficial from the OpenGM data structures that distinguishes between functions and factors, e.g. for the photo montage models of the MRF-benchmark [34]. Here, the OpenGM view is more efficient because the repeated use of the same function with multiple factors reduces cache misses.

The access to MRF-LIB is via a the single class template opengm::external::MRFLIB. Important parameters are:

- The inference algorithm inferenceType_. Possible choices are ICM, EXPANSION, SWAP, MAXPRODBP,TRWS, and BPS.
- The type of energy function energyType_. Choices are an OpenGM view VIEW, a single table for each factor TABLES, truncated linear label differences TL1, truncated quadratic label differences TL2, or a single table weighted independently for each pairwise factor WEIGHTEDTABLE.
- Maximum number of iterations numberOfIterations_
- For TRW-S, OpenGM stops if fabs(value - bound) / max(fabs(value), 1) < trwsTolerance_

3.14.2.1 ICM

```
#include <stdlib.h>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>
#include <opengm/inference/external/mrflib.hxx>

typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::external::MRFLIB<GraphicalModelType> MRFLIB;

GraphicalModelType gm;
// ...
MRFLIB::Parameter para;
para.inferenceType_ = MRFLIB::Parameter::ICM;
para.energyType_ = MRFLIB::Parameter::WEIGHTEDTABLE;
para.numberOfIterations_ = 10;

MRFLIB mrf(gm,para);
mrf.infer();

std::cout << "value: " << mrf.value() << std::endl;
```

3.14.2.2 α-Expansion

```
#include <stdlib.h>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>
#include <opengm/inference/external/mrflib.hxx>

typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::external::MRFLIB<GraphicalModelType> MRFLIB;

GraphicalModelType gm;
// ...
MRFLIB::Parameter para;
para.inferenceType_ = MRFLIB::Parameter::EXPANSION;
para.energyType_ = MRFLIB::Parameter::TL1;
para.numberOfIterations_ = 10;

MRFLIB mrf(gm,para);
mrf.infer();

std::cout << "value: " << mrf.value() << std::endl;
```

3.14.2.3 $\alpha\beta$-Swap

```
#include <stdlib.h>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>
#include <opengm/inference/external/mrflib.hxx>

typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::external::MRFLIB<GraphicalModelType> MRFLIB;

GraphicalModelType gm;
// ...
MRFLIB::Parameter para;
para.inferenceType_ = MRFLIB::Parameter::SWAP;
para.energyType_ = MRFLIB::Parameter::TL1;
para.numberOfIterations_ = 10;
```

```
MRFLIB mrf(gm,para);
mrf.infer();

std::cout << "value:␣" << mrf.value() << std::endl;
```

3.14.2.4 LBP

```
#include <stdlib.h>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>
#include <opengm/inference/external/mrflib.hxx>

typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::external::MRFLIB<GraphicalModelType> MRFLIB;

GraphicalModelType gm;
// ...
MRFLIB::Parameter para;
para.inferenceType_ = MRFLIB::Parameter::MAXPRODBP;
para.energyType_ = MRFLIB::Parameter::TABLES;
para.numberOfIterations_ = 10;

MRFLIB mrf(gm,para);
mrf.infer();

std::cout << "value:␣" << mrf.value() << std::endl;
```

3.14.2.5 BP-S

```
#include <stdlib.h>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>
#include <opengm/inference/external/mrflib.hxx>

typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::external::MRFLIB<GraphicalModelType> MRFLIB;

GraphicalModelType gm;
// ...
MRFLIB::Parameter para;
para.inferenceType_ = MRFLIB::Parameter::BPS;
para.energyType_ = MRFLIB::Parameter::VIEW;
para.numberOfIterations_ = 10;

MRFLIB mrf(gm,para);
mrf.infer();

std::cout << "value:␣" << mrf.value() << std::endl;
```

3.14.2.6 TRW-S

```
#include <stdlib.h>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/operations/minimizer.hxx>
#include <opengm/inference/external/mrflib.hxx>

typedef opengm::GraphicalModel<double, opengm::Adder> GraphicalModelType;
typedef opengm::external::MRFLIB<GraphicalModelType> MRFLIB;
```

```
GraphicalModelType gm;
// ...
MRFLIB::Parameter para;
para.inferenceType_ = MRFLIB::Parameter::TRWS;
para.energyType_ = MRFLIB::Parameter::VIEW;
para.numberOfIterations_ = 10;
para.trwsTolerance_ = 1

MRFLIB mrf(gm,para);
mrf.infer();

std::cout << "value: " << mrf.value() << " bound: " << mrf.bound() << std::endl;
```

Chapter 4

Command Line Tools

4.1 Performing Inference with Command Line Tools

The OpenGM command line interface provides a simple way to perform inference in the min-sum and max-product semi-ring. This interface is split up into several tools, each corresponding to a graphical model type and suitable inference algorithms[1].

As described also in Section 2.2, the type of a graphical model is defined by: (i) *a value type*, (ii) *an operator type*, (iii) *a set of function types*, and (iv) *a space type*.

The type of an inference algorithm is defined by: (i) *a graphical model type*, and (ii) *an accumulation type* (Chapter 3).

An command line tool can be built for each reasonable combination. Since the set of possible combinations is large, the original source includes only a few examples. Is is discussed in Section 4.2 how one can add more combinations. The set of function types is set to the same default function types which are supported even by the most basic OpenGM build: (i) *ExplicitFunction*, (ii) *PottsFunction*, (iii) *PottsNFunction*, (iv) *PottsGFunction*, (v) *TruncatedSquaredDifferenceFunction*, (vi) *TruncatedAbsoluteDifferenceFunction*. Adding and removing functions is easy (Section 4.2). The

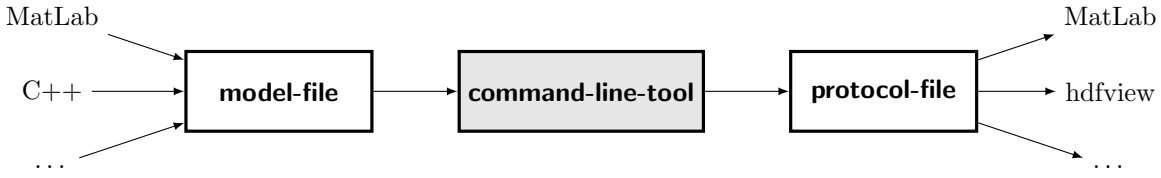


Figure 4.1: To use the opengm command line tool, one should proceed in the following: First generate the model with MatLab, in C, or any other way, than use the command line tool to generate a protocol file. This can be viewed by hdfview or loaded into MatLab, C, or the programming language of your choice.

basic work-flow for using the command line tool is sketched in Fig. 4.1. After a graphical model is generated in C++ or MATLAB, a command line tool can be used to perform inference. The command line tool can protocol the inference procedure and save optimal configurations, bounds and execution times and dump these into an HDF5 file. This file can be inspected using hdfview, read by MATLAB and C++ code or any other program that supports HDF5. Parameters of the command line tool specify parameters of inference algorithms. Building a graphical model, writing it to an HDF5 file and then performing inference by means of a command line tool can be an appealing workflow for experimental research. It makes it easy to distribute inference tasks

[1]Note that a single executable would become inconveniently large due to the various combinations of graphical models and inference algorithms

for multiple graphical models to the nodes of a cluster computer and encourages the exchange of graphical models.

NOTE: Command line tools should be built in release mode (set CMAKE_BUILD_TYPE to RELEASE and compile again) to run at peak performance.

4.1.1 Basic Usage

No matter which executable is chosen to perform inference, the command line syntax is always the same. A good starting point is the parameter *-h* or *--help* which can be used to show top level options:

```
> opengm_min_sum -h
This is the help for the commandline interface of the opengm library
Usage: opengm [arguments]
arguments:
  short name  long name          needed  description
  -h          --help             no      used to activate help output
  -v          --verbose          no      used to activate verbose output
              --modelinfo        no      used to print detailed informations about the
                  specified model
  -m          --model            yes     used to specify the desired model. Usage: filename
      :dataset
  -a          --algorithm        no      used to specify the desired algorithm
                                         permitted values: GRAPHCUT; ALPHAEXPANSION;
                                              ALPHABETASWAP; QPBO; LPCPLEX; TRWS; MRF;
                                              BELIEFPROPAGATION; TRBP; ASTAR; LAZYFLIPPER;
  -o          --outputfile       no      used to specify the desired outputfile for the
      computed results
                                         default value: PRINT ON SCREEN
  -e          --evaluate         no      used to evaluate the selected model with the
      specified input file
```

Some options, when selected, enable more specific options. These need to be set after the top level option from which they must be separated by a space. These are all top level parameters:

-h \ --help Prints the help on screen. If used in combination with parameter -a the help for the corresponding algorithm will be printed.

-v \ --verbose If used in combination with parameter -p (a standard parameter for every algorithm) verbose information on every iteration will be printed on screen (see 4.1.2 for details).

--modelinfo Prints some information about a model specified with parameter -m.

-m \ --model Used to load a model. Use filename:dataset to load a model from the dataset of a given hdf5 file.

-a \ --algorithm Specify the algorithm which will be used to perorm inference on a given model.

-o \ --outputfile Specify a name for the file in which the computed results will be stored. Depending on the file extension the results will be stored in a text file with comma separated values (.txt) or in a hdf5 file (.h5).

-e \ --evaluate Evaluates a model specified with the parameter -m with the vector stored in filename:dataset.

Each inference algorithm provides its own set of control parameters. These can be looked up by using the option *-h* in combination with the option *-a*, e.g.

```
> opengm_min_sum -h -a ALPHAEXPANSION
Printing Help for inference caller ALPHAEXPANSION
Description:
detailed description of AlphaExpansion caller...
arguments:
  short name  long name           needed  description
```

```
-p          --protocolate       no      used to enable protocolation mode. Usage: "-p N"
      where every Nth iteration step will be protocoled. If N = 0 only the final results
      will be protocoled.
                                        default value: 0
            --scale             no      Add description for scale here!!!!.
                                        default value: 1
-mf         --maxflow           no      Add description for MinSTCut here!!!!.
                                        permitted values: KOLMOGOROV; BOOST_KOLMOGOROV;
                                            BOOST_PUSH_RELABEL; BOOST_EDMONDS_KARP;
                                        default value: KOLMOGOROV
            --maxIt             no      Maximum number of iterations.
                                        default value: 1000
            --labelInitialType  no      select the desired initial label
                                        permitted values: DEFAULT; RANDOM; LOCALOPT;
                                            EXPLICIT;
                                        default value: DEFAULT
            --orderType         no      select the desired order
                                        permitted values: DEFAULT; RANDOM; EXPLICIT;
                                        default value: DEFAULT
            --randSeedOrder     no      Add description for randSeedOrder here!!!!.
                                        default value: 0
            --randSeedLabel     no      Add description for randSeedLabel here!!!!.
                                        default value: 0
            --labelorder        no      location of the file containing a vector which
                specifies the desired label order
            --label             no      location of the file containing a vector which
                specifies the desired label
```

As an example, we call the command line tool opengm_min_sum, the *α-expansion* inference algorithm for a model stored in the file MyModel.h5, in the HDF5 dataset gm1:

```
> opengm_min_sum -m MyModel.h5:gm1 -a ALPHAEXPANSION
```

The inference algorithms shipped with OpenGM have a complete set of default parameters. Thus, all parameters are optional. Each algorithm can be called by simply passing its name and a model to a command line tool. The order in which parameters are provided is irrelevant. A more complex example can be found in Section 4.1.3.

4.1.2 Keeping a Protocol

A special parameter is the parameter *-p* or *--protocolate* which is available for all inference algorithms. If this parameter is set to a value $N > 0$ then protocolation mode will be enabled which means that for every Nth iteration the current value, bound and passed time will be stored in addition to the computed optimal state. If the verbose parameter is used in combination with the protocolate parameter then the values for every Nth iteration will also be printed as console output.

4.1.3 Example for Usage of the Command Line Tool

We have converted models of the MRF-Benchmark (http://vision.middlebury.edu/MRF/results/) to the OpenGM model format and now want to use the tsukuba model to show some of the capabilities of the command line tools. The model is stored in the file *tsu-gm.h5* in the dataset *gm*.

First of all we are interested in some basic information about our model, so we run

```
> opengm_min_sum -m tsu-gm.h5:gm --modelinfo
```

which produces the following output:

```
loading model: tsu-gm.h5
using dataset: gm
Model Info
```

```
----------

Number of Variables   : 110592
Number of States      : 16
Model-Order           : 2
Number of Factors     : 331104
Number of Function Types   : 6
Acyclic Model         : 0
Chain Model           : 0
Grid Model            : 1
```

This tells us that our model consist of 110592 variables with a number of 16 states for each variable. The maximum order of a factor is 2 with a total number of 331104 factors. The model type includes 6 function types. Our model is not acyclic, has not a chain structure but a grid structure.

In the second place we want to run our model with the mrf inference algorithm to compare our computed results with the results presented on the MRF-Benchmark website. Therefore we run the following command:

```
> opengm_min_sum -a MRF -m tsu-gm.h5:gm --inference TRWS --energy TL1 -o tsu-gm-results.
    txt -p 1 -v --maxIt 5
```

Thus the inference algorithm mrf is called with our model and uses the implementation of the trws algorithm with the energy form truncated linear. Furthermore we select the file tsu-gm-results.h5 to store the computed results and protocolate each iteration step. To not blast the limits of this document we set a maximum iteration number of 5. The verbose option is set to also print all computed results on screen and the maximum number of iterations is set to 5 to not blow up the output. This leads to the following output:

```
loading model: tsu-gm.h5
using dataset: gm
running MRF caller
T: 2
Begin :        0  Value : 5.046306e+06  Bound : 1.546425e-321
Step  :        0  Value : 4.306390e+05  Bound : 3.322034e+05
Step  :        1  Value : 4.075980e+05  Bound : 3.457036e+05
Step  :        2  Value : 3.996550e+05  Bound : 3.514903e+05
Step  :        3  Value : 3.958940e+05  Bound : 3.548906e+05
Step  :        4  Value : 3.934340e+05  Bound : 3.571302e+05
End   :        5  Value : 3.934340e+05  Bound : 3.571302e+05
storing values in file: tsu-gm-results.h5
storing bounds in file: tsu-gm-results.h5
storing times in file: tsu-gm-results.h5
storing corresponding iterations in file: tsu-gm-results.h5
storing optimal states in file: tsu-gm-results.h5
```

The output shows some IO information, and more interesting that our energy function is truncated at 2 and the values and bounds computed at each iteration step these correspond to the results stored in tsu-gm-results.h5 as shown in Figure 4.2 and are also identical to those values presented on the MRF website.

Finally after we have computed the optimal state we can now check if the optimal state coincident with the computed optimal value. This can be done by using the parameter *--evaluate* which gives the following result:

```
> opengm_min_sum -m tsu-gm.h5:gm --evaluate tsu-gm-results.h5:states
loading model: tsu-gm.h5
using dataset: gm
result: 3.934340e+05
```

This is the same value as the one we got from our inference run above. So our optimal state seems to coincident with our computed optimal value.

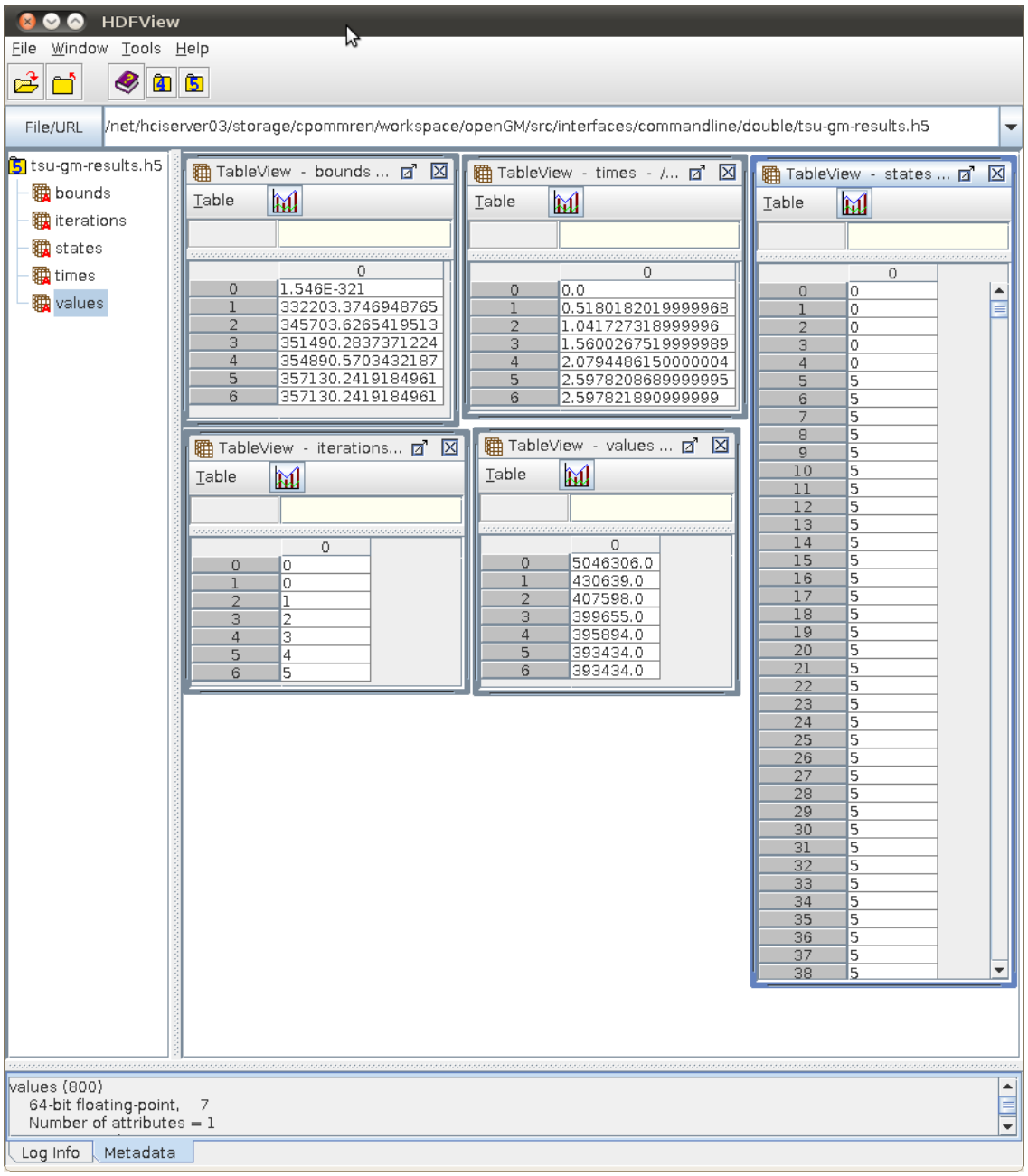


Figure 4.2: Results of the MRF-Benchmark tsukuba computed with OpenGM command line tools with the call *opengm_min_sum -a MRF -m tsu-gm.h5:gm –inference TRWS –energy TL1 -o tsu-gm-results.txt -p 1 -v –maxIt 5*

4.1.4 Reading Protocol Files in MatLab

Reading protocol files in MatLab is very simple. After the directory including the additional MatLab scripts has been added to the MatLab path by

```
addpath('~/opengm2/matlab')
```

one can use the functions marray_load and marray_save to load and save multidimensional arrays into HDF5-files, respectively. The following code load vectors of values, bounds and times stored by the command line tool as well as the final state-vector.

```
addpath('~/opengm2/matlab')

resfile = './tsu-gm-results.h5';
times   = marray_load(resfile,'times');
bounds  = marray_load(resfile,'bounds');
values  = marray_load(resfile,'values');

statevector = marray_load('out-opengm_lbp-gm94.h5','states');
```

4.2 Changing Model Type

If you want to use the OpenGM command line tools with our own model, functions, or types, you have to create a new executable which supports this model type. The best way to do this is to look at in the OpenGM source folder (src/interfaces/commandline/). There are several example codes for different types of models.

For instance the minimalistic code to create an executable which only supports a graphical model with Value Type *float*, Index Type *unsigned int*, Label Type *unsigned char*, Operator Type *opengm::Adder*, and Function Type *opengm::ExplicitFunction* and the bruteforce inference algorithm with Accumulation Type *opengm::Minimizer* is as follows:

```
#include <iostream>

// opengGM stuff
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/graphicalmodel/graphicalmodel_hdf5.hxx>
#include <opengm/operations/minimizer.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/functions/explicit_function.hxx>

// commandline IO interface
#include "opengm/src/interfaces/commandline/cmd_interface.hxx"

// inference caller
#include "opengm/src/interfaces/common/caller/bruteforce_caller.hxx"

int main(int argc, char** argv) {
   if(argc < 2) {
      std::cerr << "At least one input argument required" << std::endl;
      std::cerr << "try \"-h\" for help" << std::endl;
      return 1;
   }

   typedef float ValueType;
   typedef unsigned int IndexType;
   typedef unsigned char LabelType;
   typedef opengm::Adder OperatorType;
   typedef opengm::Minimizer AccType;
   typedef opengm::interface::IOCMD InterfaceType;
   typedef DiscreteSpace<IndexType, LabelType> SpaceType;
```

```
                                                                                    31
    // Set functions for graphical model                                            32
    typedef opengm::meta::TypeListGenerator<                                        33
        opengm::ExplicitFunction<ValueType>                                         34
    >::type FunctionTypeList;                                                       35
                                                                                    36
                                                                                    37
    typedef opengm::GraphicalModel<                                                 38
        ValueType,                                                                  39
        OperatorType,                                                               40
        FunctionTypeList,                                                           41
        SpaceType                                                                   42
    > GmType;                                                                       43
                                                                                    44
    typedef opengm::meta::TypeListGenerator <                                       45
        opengm::interface::BruteforceCaller<interface::IOCMD, GmType, AccType>      46
    >::type InferenceTypeList;                                                      47
                                                                                    48
    interface::CMDInterface<GmType, InferenceTypeList> interface(argc, argv);       49
    interface.parse();                                                              50
                                                                                    51
    return 0;                                                                       52
}                                                                                   53
```

To change the desired OpenGM Model Type only the lines 23-26 and 29-44 have to be touched.

4.3 Adding Inference Methods

To add the support for a new inference method to a command line interface two things have to be done:

1. Write a *Caller Class* for the inference method.
2. Add the caller to the inference type list in the command line tool.

Write a Caller Class for the Inference Method:

Say you have followed the guide of section 6.4 and implemented a new inference algorithm called *mySolver*. Then the code for the corresponding caller looks as follows:

```
#include <opengm/opengm.hxx>                                                        1
                                                                                    2
// implementation of mySolver                                                       3
#include <opengm/inference/mySolver.hxx>                                            4
                                                                                    5
// caller base class                                                                6
#include "inference_caller_base.hxx"                                                7
                                                                                    8
// command line arguments                                                           9
#include "../argument/argument.hxx"                                                 10
                                                                                    11
namespace opengm {                                                                  12
namespace interface {                                                               13
                                                                                    14
template <class IO, class GM, class ACC>                                            15
class mySolverCaller : public InferenceCallerBase<IO, GM, ACC> {                    16
protected:                                                                          17
    using InferenceCallerBase<IO, GM, ACC>::io_;                                    18
    using InferenceCallerBase<IO, GM, ACC>::infer;                                  19
                                                                                    20
    typedef typename opengm::mySolver<GM, ACC> mySolver;                            21
    typename mySolver::Parameter mySolverParameter_;                                22
    typedef typename mySolver::TimingVisitorType TimingVisitorType;                 23
```

```
    virtual void
    runImpl(GM& model, StringArgument<>& outputfile, const bool verbose);
public:
    const static std::string name_;
    mySolverCaller(IO& ioIn);
};

template <class IO, class GM, class ACC>
inline mySolverCaller<IO, GM, ACC>::ICMCaller(IO& ioIn)
    : InferenceCallerBase<IO, GM, ACC>("MYSOLVER",
            "detailed description of mySolver Parser...", ioIn) {
    // add commandline arguments required by mySolver here...
}

template <class IO, class GM, class ACC>
inline void
mySolverCaller<IO, GM, ACC>::runImpl(GM& model,
        StringArgument<>& outputfile, const bool verbose) {
    std::cout << "running mySolver caller" << std::endl;

    // add code for setting up mySolver here...

    // run inference and protocolate results
    this-> template infer<ICM, TimingVisitorType,
        typename mySolverCaller::Parameter>(model, outputfile, verbose,
                                            mySolverParameter_);
}

template <class IO, class GM, class ACC>
const std::string mySolverCaller<IO, GM, ACC>::name_ = "MYSOLVER";

} // namespace interface
} // namespace opengm
```

The constructor sets up all necessary information for the command line tools. If you want to add command line parameters which might be required by your solver, this is the place to go (see chapter 4.4 for details).

The function *runImpl()* will be called by the command line tools if a user requests inference with your solver. If your solver requires some setup before *mySolver::infer()* is called, this has to be done in the *runImpl()*-function before the call of the base class function *InferenceCallerBase::infer()*. This base class function takes care of the right call of *mySolver::infer()* and handles storing all user requested values in the specified output file. More examples for existing inference algorithms an be found in /src/interfaces/common/caller/.

Add the Caller to the Inference Type List:

After you have implemented the caller, you have to add it to the inference type list of the command line tool which you want to use. The following code shows the example of chapter 4.2 for a minimalistic command line executable. This time extended by our new caller.

```
#include <iostream>

// opengGM stuff
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/graphicalmodel/graphicalmodel_hdf5.hxx>
#include <opengm/operations/minimizer.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/functions/explicit_function.hxx>

// commandline IO interface
#include "opengm/src/interfaces/commandline/cmd_interface.hxx"
```

```
// inference caller
#include "opengm/src/interfaces/common/caller/bruteforce_caller.hxx"

// our new written mySolver caller
#include "mysolver_caller.hxx"

int main(int argc, char** argv) {
   if(argc < 2) {
      std::cerr << "At least one input argument required" << std::endl;
      std::cerr << "try \"-h\" for help" << std::endl;
      return 1;
   }

   typedef float ValueType;
   typedef unsigned int IndexType;
   typedef unsigned char LabelType;
   typedef opengm::Adder OperatorType;
   typedef opengm::Minimizer AccType;
   typedef opengm::interface::IOCMD InterfaceType;
   typedef DiscreteSpace<IndexType, LabelType> SpaceType;

   // Set functions for graphical model
   typedef opengm::meta::TypeListGenerator<
      opengm::ExplicitFunction<ValueType>
   >::type FunctionTypeList;


   typedef opengm::GraphicalModel<
      ValueType,
      OperatorType,
      FunctionTypeList,
      SpaceType
   > GmType;

   typedef opengm::meta::TypeListGenerator <
      opengm::interface::BruteforceCaller<interface::IOCMD, GmType, AccType>,
      opengm::interface::mySolverCaller<interface::IOCMD, GmType, AccType>
   >::type InferenceTypeList;

   interface::CMDInterface<GmType, InferenceTypeList> interface(argc, argv);
   interface.parse();

   return 0;
}
```

All callers which should be supported by a command line toll have to be added to the *Inference-TypeList*.

4.4 Adding Parameters

Following the example of a basic caller in chapter 4.3 we will now see how to add command line paramters to the caller to allow the user to specify the behaviour of mySolver.

There are two types of paramters which are supported by the OpenGM command line tools.

First, the *flag-parameter* which can be used to turn the corresponding behaviour *on* or *off*. This type of parameter does not require any further user input as it is basically a bool variable set to either true or false and that why called *BoolArgument*. BoolArgument has only one constructor with the signature (bool& storageIn, const std::string& shortNameIn, const std::string& longNameIn, const std::string& descriptionIn) which is identical to the first constructor of the list above, except the requiredIn value as it makes no sense to enforce a flag to be set.

Second, the *value-parameter* which can take a user input to set its value. This type of parameter might have a default value, in which case no further user input is required or it can be tagged as required. In this case the user has to enter a value which has to be entered after the parameter and

has to be separated by a space. Additionally a parameter might have a list of permitted values. If so, the execution of the command line interface will be aborted if the user doesn't enter a value which is part of the permitted values.

The following value-paramter types are supported: (i) *FloatArgument*, (ii) *DoubleArgument*, (iii) *IntArgument*, (iv) *Size_TArgument*, (v) *StringArgument*, (vi) *VectorArgument*.

All arguments except BoolArgument have four different constructors which allow to specify the behaviour of the argument.

The first constructor has the signature (T& storageIn, const std::string& shortNameIn, const std::string& longNameIn, const std::string& descriptionIn, const bool requiredIn = false). Where storageIn is a reference to the variable in which the user input shall be stored, shortNameIn is the short cut name of the parameter (if an empty string is passed as shortNameIn, the user can only use the long name to set the parameter), longNameIn is the long name of the parameter, descriptionIn is the description of the parameter which will be printed to the command line tools user if he calls for help and requiredIn can be used to force the command line tools user to set a value for this parameter.

The second constructor has the signature (T& storageIn, const std::string& shortNameIn, const std::string& longNameIn, const std::string& descriptionIn, const T& defaultValueIn) and is identical to the first constructor except that requiredIn is exchanged with defaultValueIn to allow the usage of a default value if the command line tools user doesn't set a value for this parameter.

The third constructor has the signature (T& storageIn, const std::string& shortNameIn, const std::string& longNameIn, const std::string& descriptionIn, const bool requiredIn, const CONTAINER& permittedValuesIn) and is an expansion of the first constructor extended by the parameter permittedValuesIn which is used to provide a container containing all allowed values for the argument.

The fourth constructor has the signature (T& storageIn, const std::string& shortNameIn, const std::string& longNameIn, const std::string& descriptionIn, const T& defaultValueIn, const CONTAINER& permittedValuesIn) and is an expansion of the second constructor extended by the parameter permittedValuesIn which is used to provide a container containing all allowed values for the argument.

Say you have implemented mySolver in a way that the user has the possibility to influence the behaviour of mySolver. For instance, the parameter of mySolver might have the following options to be set: (i) *x0* (A possible starting point represented by a vector of matching dimension), (ii) *fastMath* (A flag to activate faster mathematic computations), (iii) *eps* (A double value representing the allowed tolerance), (iv) *maxIter* (An integer value representing the maximum number of iterations), (v) *heuristic* (A switch to select the desired heuristic. Allowed values are: DefultHeuristic, FastHeuristic or StrictHeuristic). Then the caller for mySolver with respect to the possible paramter would look as follows:

```
#include <opengm/opengm.hxx>                                                    1
                                                                                2
// implementation of mySolver                                                   3
#include <opengm/inference/mySolver.hxx>                                        4
                                                                                5
// caller base class                                                            6
#include "inference_caller_base.hxx"                                            7
                                                                                8
// command line arguments                                                       9
#include "../argument/argument.hxx"                                             10
                                                                                11
namespace opengm {                                                              12
namespace interface {                                                           13
                                                                                14
template <class IO, class GM, class ACC>                                        15
class mySolverCaller : public InferenceCallerBase<IO, GM, ACC> {                16
protected:                                                                      17
   using InferenceCallerBase<IO, GM, ACC>::io_;                                 18
   using InferenceCallerBase<IO, GM, ACC>::infer;                               19
```

```
    typedef typename opengm::mySolver<GM, ACC> mySolver;
    typename mySolver::Parameter mySolverParameter_;
    typedef typename mySolver::TimingVisitorType TimingVisitorType;
    virtual void
    runImpl(GM& model, StringArgument<>& outputfile, const bool verbose);

    std::string selectedHeuristic_;

public:
    const static std::string name_;
    mySolverCaller(IO& ioIn);
};

template <class IO, class GM, class ACC>
inline mySolverCaller<IO, GM, ACC>::ICMCaller(IO& ioIn)
    : InferenceCallerBase<IO, GM, ACC>("MYSOLVER",
            "detailed_description_of_mySolver_Parser...", ioIn) {
    // add commandline arguments required by mySolver here...
    addArgument(VectorArgument<std::vector<size_t> >(mySolverParameter_.startPoint_,"x0",
      "startingpoint", "location_of_the_file_containing_the_values_for_the_starting_point", false));
    addArgument(BoolArgument(mySolverParameter_.fastMath_, "", "fastMath",
      "Allow_functions_to_be_shared"));
    addArgument(DoubleArgument<>(mySolverParameter_.eps_, "", "eps",
      "allowed_tolerance", mySolverParameter_.eps));
    addArgument(Size_TArgument<>(mySolverParameter_.numberOfIterations_, "", "maxIter",
      "Maximum_number_of_iterations.", mySolverParameter_.numberOfIterations_));

    std::vector<std::string> possibleHeuristics;
    possibleHeuristics.push_back("DefultHeuristic");
    possibleHeuristics.push_back("FastHeuristic");
    possibleHeuristics.push_back("StrictHeuristic");
    addArgument(StringArgument<>(selectedHeuristic_, "", "heuristic",
      "Select_desired_heuristic.", possibleHeuristics.front(), possibleHeuristics));
}

template <class IO, class GM, class ACC>
inline void
mySolverCaller<IO, GM, ACC>::runImpl(GM& model,
        StringArgument<>& outputfile, const bool verbose) {
    std::cout << "running_mySolver_caller" << std::endl;

    // add code for setting up mySolver here...
    // transform string into corresponding parameter value
    if(selectedHeuristic_ == "DefultHeuristic") {
        mySolverParameter_.heuristic_= mySolver::Parameter::DEFAULT;
    } else if(selectedInferenceType_ == "FastHeuristic") {
        mySolverParameter_.heuristic_= mySolver::Parameter::FAST;
    } else if(selectedInferenceType_ == "StrictHeuristic") {
        mySolverParameter_.heuristic_= mySolver::Parameter::STRICT;
    } else {
      throw RuntimeError("Unknown_inference_type_for_mySolver");
    }

    // run inference and protocolate results
    this-> template infer<ICM, TimingVisitorType,
        typename mySolverCaller::Parameter>(model, outputfile, verbose,
                                            mySolverParameter_);
}

template <class IO, class GM, class ACC>
const std::string mySolverCaller<IO, GM, ACC>::name_ = "MYSOLVER";

} // namespace interface
} // namespace opengm
```

4.5 Adding Return Types

If protocol mode is active, the results computed by an inference algorithm called by the command line tools are stored using the default timing visitor. This is done in the template method *protocolate()* of the base class *InferenceCallerBase*. If you want to get more information out of your inference call, you have to specialize the visitors or write a new visitor which logs the desired information. Further more the template method *protocolate()* has to be specialized for the new visitor type. The base template method should provide a sufficient example on how the specialized method is supposed to look like.

Chapter 5

OpenGM and MATLAB

OpenGM does not come with a full-fletched MATLAB interface. It does, however, contain the C++ code for a MATLAB mex file to construct OpenGM models within MATLAB and save these in the native OpenGM file format. The file format can be read by any OpenGM C++ code, in particular by OpenGM's command line tools. These write results to an HDF5 file that can be read back into MATLAB.

The code below is for a MATLAB mex file that builds second order models. It can be called in MATLAB with matrices first_order_functions and pairs and a cell-array of matrices second_order_functions as opengmBuild(first_order_functions, pairs, second_order_functions, 'model.h5'). The reader will find it easily to extended this code to build more complex models.

```
#include "mex.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <vector>

#include <opengm/operations/adder.hxx>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/graphicalmodel/graphicalmodel_hdf5.hxx>
#include <opengm/utilities/metaprogramming.hxx>

typedef double ValueType;
typedef size_t IndexType;
typedef unsigned char LabelType;
typedef opengm::GraphicalModel<
   ValueType,
   opengm::Adder,
   opengm::meta::TypeListGenerator<opengm::ExplicitFunction<ValueType> >::type,
   opengm::DiscreteSpace<IndexType, LabelType>,
   false
   > GraphicalModelType;

typedef opengm::ExplicitFunction<ValueType> ExplicitFunctionType;
typedef GraphicalModelType::FunctionIdentifier FunctionIdentifier;

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
   /* INPUT */
   /* 1.) unaray*/
   /* 2.) pairs*/
   /* 3.) pairfunction*/
   /* 4.) output-filename */

   if (nrhs != 4) {
      mexErrMsgTxt("Incorrect number of inputs.");
```

```
    }

    char* filename;
    filename = (char*)mxCalloc(mxGetN(prhs[3])+1,sizeof(char));
    mxGetString(prhs[3], filename, mxGetN(prhs[3])+1);

    double* unary = (double*) mxGetData(prhs[0]);
    double* pairwise_node = (double*) mxGetData(prhs[1]);
    size_t numberOfVariables = mxGetN(prhs[0]);
    size_t numberOfFactors = mxGetN(prhs[1]);

    /* Output Information*/
    printf("Number of variables:          %d\n",numberOfVariables);
    printf("Number of pairwise factors:    %d\n",numberOfFactors);
    printf("Output file:                   %s\n",filename);

    std::vector<LabelType> numStates(numberOfVariables,numberOfVariables);
    GraphicalModelType gm(
        opengm::DiscreteSpace<IndexType, LabelType >(numStates.begin(), numStates.end())
    );

    //Add Unaries
    std::cout << "add_Unaries"<<std::endl;
    for(size_t var=0; var<numberOfVariables; ++var) {
        LabelType shape[] = {gm.numberOfLabels(var)};
        ExplicitFunctionType func(shape,shape+1);
        for(size_t i=0; i<shape[0]; ++i) {
            func(i) = unary[i+var*numberOfVariables];
        }
        FunctionIdentifier fid = gm.addSharedFunction(func);
        gm.addFactor(fid, &var, &var+1);
    }

    //ADD Pairwise
    std::cout << "add_pairwise"<<std::endl;
    for(size_t f=0;f<numberOfFactors;++f) {
        IndexType vars[2];
        LabelType shape[]={numberOfVariables,numberOfVariables};
        vars[0] = pairwise_node[0+2*f];
        vars[1] = pairwise_node[1+2*f];
        ExplicitFunctionType func(shape,shape+2);
        mxArray* data = mxGetCell(prhs[2], f);
        double* pdata = (double*) mxGetData(data);
        for(size_t i1=0; i1<shape[0]; ++i1) {
            for(size_t i2=0; i2<shape[1]; ++i2) {
                func(i1,i2) = pdata[i1+i2*shape[0]];
            }
        }
        FunctionIdentifier fid = gm.addSharedFunction(func);
        gm.addFactor(fid, vars, vars+2);
    }

    opengm::hdf5::save(gm, filename, "gm");
    mxFree(filename);
}
```

Chapter 6

Extending OpenGM

6.1 Label Spaces

In special cases, neither of the label spaces described in Section 2.3 is maximally efficient. As an example, consider models in which one special variable attains a large number of labels while all other variables are binary. OpenGM accounts for such special cases by allowing the user to write their own label spaces. The custom class template needs to inherit from SpaceBase:

```
#include "opengm/opengm.hxx"
#include "opengm/graphicalmodel/space/space_base.hxx"

template<class I = std::size_t, class L = std::size_t>
class CustomSpace : public SpaceBase<CustomSpace<I, L>, I, L> {
public:
   typedef I IndexType;
   typedef L LabelType;

   CustomSpace();
   CustomSpace(const IndexType labels, const IndexType variables)
      : numberOfLabelsOfFirstVariable_(labels),
        numberOfVariables_(variables)
      { OPENGM_ASSERT(labels > 0);
        OPENGM_ASSERT(variables > 1); }
   void assign(const IndexType labels, const IndexType variables)
      { OPENGM_ASSERT(labels > 0);
        OPENGM_ASSERT(variables > 1);
        numberOfLabelsOfFirstVariable_ = labels;
        numberOfVariables_ = variables; }
   IndexType addVariable(const IndexType labels)
      { OPENGM_ASSERT(labels == 2);
        ++numberOfVariables_; }
   IndexType numberOfVariables() const
      { return numberOfVariables_; }
   LabelType numberOfLabels(const IndexType variable) const
      { return variable == 0 ? numberOfLabelsOfFirstVariable_ : 2; }

private:
   IndexType numberOfLabelsOfFirstVariable_;
   IndexType numberOfVariables_;
};
```

6.2 Functions

In principle, every graphical model function in OpenGM can be implemented by its value table, using the class ExplicitFunction. However, as described in Section 2.4, this is often not the most

efficient representation. Therefore, OpenGM allows the user to write their own class templates for functions.

As an example, consider the modulo function $f(x) = x \bmod 2$ that yields 0, if x is even, and 1, if x is odd. Implementing this function by its value table would surely be a waste of memory if x can assume many different labels. A class template for the modulo function, as well as any custom function class template, needs to inherit from FunctionBase:

```
#include "opengm/opengm.hxx"
#include "opengm/functions/function_registration.hxx"
#include "opengm/functions/function_properties_base.hxx"

template<class T, class I = size_t, class L = size_t>
class ModuloFunction
: public FunctionBase<ModuloFunction<T, I, L>, T, size_t, size_t>
{
public:
   typedef T ValueType;
   typedef L LabelType;
   typedef I IndexType;

   ModuloFunction(const IndexType shape)
      : shape_(shape); {}
   LabelType shape(const size_t j) const
      { OPENGM_ASSERT(j == 0); return shape_; }
   size_t size() const
      { return shape_; }
   size_t dimension() const
      { return 1; }
   template<class ITERATOR> ValueType operator()(ITERATOR it) const
      { OPENGM_ASSERT(*it < shape_); return *it % 2; }
   bool operator==(const ModuloFunction& other) const
      { return other.shape_ == shape_; }
   IndexType numberOfParameters() const
      { return 1; }
   IndexType parameter(const size_t j) const
      { OPENGM_ASSERT(j == 0); return shape_; }
   IndexType& parameter(const size_t j);
      { OPENGM_ASSERT(j == 0); return shape_; }

private:
   IndexType shape_;

friend class FunctionSerialization<ModuloFunction<T, I, L> > ;
};
```

Every custom function needs to be registered with a unique ID between 0 and 16000 (IDs above 16000 are reserved for the OpenGM standard). Registration works as follows:

```
template <class T, class I, class L>
struct FunctionRegistration<ModuloFunction<T, I, L> > {
   enum ID { Id = 0; };
};
```

One last step is necessary to add the custom class of functions to the OpenGM file format. This works by specializing the class template FunctionSerialization:

```
template<class T, class I, class L>
class FunctionSerialization<ModuloFunction<T, I, L> > {
public:
   typedef typename ModuloFunction<T, I, L>::ValueType ValueType;

   static size_t indexSequenceSize(const ModuloFunction<T, I, L>&)
      { return 1; }
   static size_t valueSequenceSize(const ModuloFunction<T, I, L>&)
```

```
    { return 0; }
  template<class INDEX_ITERATOR, class VALUE_ITERATOR>
    static void serialize(const ModuloFunction<T, I, L>& f,
      INDEX_ITERATOR iit, VALUE_ITERATOR vit)
        { *iit = f.size(); }
  template<class INDEX_ITERATOR, class VALUE_ITERATOR>
    static void deserialize(INDEX_ITERATOR iit, VALUE_ITERATOR vit,
      ModuloFunction<T, I, L>& f)
        { f.shape_ = *iit; }
};
```

6.3 Operations

Commutative and associative operations are classes in OpenGM, equipped with member functions for the operation itself and its inverse, the neutral and inverse neutral element. Member functions for an (inverse) hyper-operation and an order associated with the operation are optional. The complete interface (for brevity without static and const qualifiers) is shown in Tab. 6.1. Operations are used in algorithms like in the example below:

T neutral<T>()	returns the neutral element
void neutral(T& out)	returns the neutral element by reference
T ineutral<T>()	returns the inverse neutral element
void ineutral(T& out)	returns the inverse neutral element by reference
void op(T1& in1, T2& out)	in-place operation (out = op(out,in1))
void op(T1& in1, T2& in2, T3& out)	operation (out =op(in1,in2))
void iop(T1& in1, T2& out)	in-place inverse operation (out = iop(out,in1))
void iop(T1& in1, T2& in2, T3& out)	inverse operation (out = iop(in1,in2))
void hop(T1& in, T2& out)	in-place hyper-operation (out = hop(out,in1))
void hop(T1& in1,T2& in2, T3& out)	hyper-operation (out =hop(in1,in2))
void ihop(T1& in, T2& out)	in-place inverse hyper-operation (out = ihop(out,in1))
void ihop(T1& in1,T2& in2, T3& out)	inverse hyper-operation (out = ihop(in1,in2))
bool hasbop()	returns true if operation provides a compare operation
bool bop(T& in1, T& in2)	boolean compare operation
bool ibop(T& in1, T& in2)	inverse boolean compare operation

Table 6.1: Interface of operations in OpenGM

```
#include <opengm/datastructures/marray/marray.hxx>
#include <opengm/operations/adder.hxx>

typedef double T;
typedef opengm::Adder OP;

std::vector<size_t> shape(2,4);
marray::Marray<T> A(shape.begin(),shape.end(),2);
marray::Marray<T> B(shape.begin(),shape.end(),6);
marray::Marray<T> C(shape.begin(),shape.end(),1);

OP::op(A,B,C); //C(.,.)=8
OP:iop(A,C); //C(.,.)=10

double d = OP::neutral<double>(); //d=0
OP::ineutral(d); //d=oo
```

6.4 Algorithms

All algorithms implemented in OpenGM access graphical models via the general interface described in Section 2.8. This interface is designed w.r.t. performance because its functions are called often,

in inner loops of algorithms[1].

Algorithms themselves share a parsimonious interface called Inference[2]. Deriving a custom algorithm from Inference has advantages, e.g. when it comes to using algorithms exchangeably or adding custom algorithms to the OpenGM command line tools (Section 4).

By far the best example of how to implement a custom algorithm is the simple brute force search, include/opengm/inference/bruteforce.hxx.

[1]Inheritance with virtual functions is avoided altogether within a graphical model.

[2]Inheritance with virtual functions in the interface class template Inference is used such that algorithms are exchangeable. The effect on the runtime is negligible because the interface of *algorithms* will almost never become a runtime bottleneck.

Chapter 7

Examples

7.1 Markov Chain

```
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/graphicalmodel/space/simplediscretespace.hxx>
#include <opengm/functions/potts.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/inference/messagepassing.hxx>

using namespace std; // 'using' is used only in example code
using namespace opengm;

int main() {
   // construct a label space with numberOfVariables many variables,
   // each having numberOfLabels many labels
   const size_t numberOfVariables = 40;
   const size_t numberOfLabels = 5;
   typedef SimpleDiscreteSpace<size_t, size_t> Space;
   Space space(numberOfVariables, numberOfLabels);

   // construct a graphical model with
   // - addition as the operation (template parameter Adder)
   // - support for Potts functions (template parameter PottsFunction<double>)
   typedef OPENGM_TYPELIST_2(ExplicitFunction<double>, PottsFunction<double>) FunctionTypelist;
   typedef GraphicalModel<double, Adder, FunctionTypelist, Space> Model;
   Model gm(space);

   // for each variable, add one 1st order functions and one 1st order factor
   for(size_t v = 0; v < numberOfVariables; ++v) {
      const size_t shape[] = {numberOfLabels};
      ExplicitFunction<double> f(shape, shape + 1);
      for(size_t s = 0; s < numberOfLabels; ++s) {
         f(s) = static_cast<double>(rand()) / RAND_MAX;
      }
      Model::FunctionIdentifier fid = gm.addFunction(f);

      size_t variableIndices[] = {v};
      gm.addFactor(fid, variableIndices, variableIndices + 1);
   }

   // add one (!) 2nd order Potts function
   PottsFunction<double> f(numberOfLabels, numberOfLabels, 0.0, 0.3);
   Model::FunctionIdentifier fid = gm.addFunction(f);

   // for each pair of consecutive variables,
   // add one factor that refers to the Potts function
   for(size_t v = 0; v < numberOfVariables - 1; ++v) {
      size_t variableIndices[] = {v, v + 1};
```

```
    gm.addFactor(fid, variableIndices, variableIndices + 2);
  }

  // set up the optimizer (loopy belief propagation)
  typedef BeliefPropagationUpdateRules<Model, Minimizer> UpdateRules;
  typedef MessagePassing<Model, Minimizer, UpdateRules, MaxDistance> BeliefPropagation;
  const size_t maxNumberOfIterations = numberOfVariables * 2;
  const double convergenceBound = 1e-7;
  const double damping = 0.0;
  BeliefPropagation::Parameter parameter(maxNumberOfIterations, convergenceBound, damping);
  BeliefPropagation bp(gm, parameter);

  // optimize (approximately)
  BeliefPropagation::VerboseVisitorType visitor;
  bp.infer(visitor);

  // obtain the (approximate) argmin
  vector<size_t> labeling(numberOfVariables);
  bp.arg(labeling);
}
```

7.2 Bipartite Matching

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>

#include <opengm/opengm.hxx>
#include <opengm/datastructures/marray/marray.hxx>
#include <opengm/functions/potts.hxx>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/inference/astar.hxx>

using namespace std; // 'using' is used only in example code
using namespace opengm;

template<class T>
void createAndPrintData(size_t nrOfVariables,marray::Marray<T> & data)
{
	size_t shape[]={nrOfVariables,nrOfVariables};
	data.resize(shape,shape+2);
	cout<<"pariwise costs:\n";
	srand(0);
	for(size_t v=0;v<data.shape(0);++v) {
		for(size_t s=0;s<data.shape(0);++s) {
			data(v,s)=static_cast<float>(rand()%100) *0.01;
			cout<<left<<setw(6)<<setprecision(2)<<data(v,s);
		}
		cout<<"\n";
	}
}
void printSolution(const vector<size_t> & solution)
{
	set<size_t> unique;
	cout<<"\nSolution States :\n";
	for(size_t v=0;v<solution.size();++v) {
		cout<<left<<setw(2)<<v<<" -> "<<solution[v]<<"\n";
	}
}
int main(int argc, char** argv)
{
  // model parameters
	const size_t nrOfVariables=5;
```

```
    const size_t nrOfStates=nrOfVariables;
float high=20;
    cout<<"\n Matching with one to one correspondences:\n"
    <<nrOfVariables <<"variables with "<<nrOfStates<<" states \n\n";
    // create data terms / pairwise costs
    marray::Marray<float> data;
    createAndPrintData(nrOfVariables,data);
    // Build the Model
    // type of the graphical model:
    typedef GraphicalModel
    <
        float,
        Adder,
        OPENGM_TYPELIST_2(PottsFunction<float>,opengm::ExplicitFunction<float>)
    >
    Model;
    typedef ExplicitFunction<float> ExplicitFunctionType ;
    typedef Model::FunctionIdentifier FunctionIdentifier;

    // construct a graphical model with
// - nrOfVariables variables,
// each having nrOfStates labels
// - addition as the operation (template parameter Adder)
// - support for Potts functions (template parameter PottsFunction<double>))

    vector<size_t> variableStates(nrOfVariables,nrOfStates);
    Model gm(DiscreteSpace<size_t,size_t> (variableStates.begin(),variableStates.end()));
    //unary potentials
    {
        //shape of the function f
        const size_t shape[]={nrOfStates};
        //construct the function
        ExplicitFunctionType f(shape,shape+1);
        //add factors and functions
        for(size_t v=0;v<nrOfVariables;++v) {
            for(size_t s=0;s<nrOfStates;++s) {
                f(s)=1.0f-data(v,s);
            }
            FunctionIdentifier id=gm.addFunction(f);
            size_t vi[]={v};
            gm.addFactor(id,vi,vi+1);
        }
    }
    //pair potentials (Potts)
    {
        // add one (!) 2nd order Potts function
        PottsFunction<float> f(nrOfStates,nrOfStates,high,0);
        FunctionIdentifier id=gm.addFunction(f);
        //add pair potentials for all variables
        for(size_t v1=0;v1<nrOfVariables;++v1)
        for(size_t v2=v1+1;v2<nrOfVariables;++v2) {
            //create the variables indices for the factor
            size_t vi[]={ v1,v2 };
            // sort the variables indices is not necessary
            // since v1 is v1 < v2 true by construction
            // but remember SORTED VARIABLE SEQUENCE IS PRECODITION
            // add factor
            gm.addFactor(id,vi,vi+2);
        }
    }
    // set up the optimizer (A*Star)
    typedef AStar<Model, Minimizer > AstarType;
    AstarType astar(gm);

    // obtain the argmin
    AstarType::VerboseVisitorType verboseVisitor;
    cout<<"\nStart Astar:\n";
```

```
    astar.infer(verboseVisitor);
    // output the argmin
    vector<size_t> solution;
    astar.arg(solution);
    printSolution(solution);
}
```

7.3 Interpixel-Boundary Segmentation

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
#include <cmath>

#include <opengm/opengm.hxx>
#include <opengm/datastructures/marray/marray.hxx>
#include <opengm/graphicalmodel/space/simplediscretespace.hxx>
#include <opengm/graphicalmodel/graphicalmodel.hxx>
#include <opengm/operations/adder.hxx>
#include <opengm/inference/lazyflipper.hxx>

using namespace std; // 'using' is used only in example code

// this class is used to map a node (x, y) in the topological
// grid to a unique variable index
class TopologicalCoordinateToIndex {
public:
   TopologicalCoordinateToIndex(
      const size_t geometricGridSizeX,
      const size_t geometricGridSizeY
   )
   : gridSizeX_(geometricGridSizeX),
      gridSizeY_(geometricGridSizeY)
   {}

   const size_t operator()(
      const size_t tx,
      const size_t ty
   ) const {
      return tx / 2 + (ty / 2)*(gridSizeX_) + ((ty + ty % 2) / 2)*(gridSizeX_ - 1);
   }

   size_t gridSizeX_;
   size_t gridSizeY_;
};

template<class T>
void randomData(
   const size_t gridSizeX,
   const size_t gridSizeY,
   marray::Marray<T>& data
) {
   srand(gridSizeX * gridSizeY);
   const size_t shape[] = {gridSizeX, gridSizeY};
   data.assign();
   data.resize(shape, shape + 2);
   for (size_t y = 0; y < gridSizeY; ++y) {
      for (size_t x = 0; x < gridSizeX; ++x) {
         data(x, y) = static_cast<float> (rand() % 10) *0.1;
      }
   }
}
```

```
template<class T>
void printData(
   const marray::Marray<T> & data
) {
   cout << "energy for boundary to be active:" << endl;
   for (size_t y = 0; y < data.shape(1)*2 - 1; ++y) {
      for (size_t x = 0; x < data.shape(0)*2 - 1; ++x) {
         if (x % 2 == 0 && y % 2 == 0) {
            cout << left << setw(3) << setprecision(1) << data(x / 2, y / 2);
         } else if (x % 2 == 0 && y % 2 == 1) {
            cout << left << setw(3) << setprecision(1) << "___";
         } else if (x % 2 == 1 && y % 2 == 0) {
            cout << left << setw(3) << setprecision(1) << " | ";
         } else if (x % 2 == 1 && y % 2 == 1) {
            cout << left << setw(3) << setprecision(1) << " * ";
         }
      }
      cout << "\n";
   }
}

// output the (approximate) argmin
template<class T>
void printSolution(
   const marray::Marray<T>& data,
   const vector<size_t>& solution
)
{
   TopologicalCoordinateToIndex cTHelper(data.shape(0), data.shape(1));
   cout << endl << "solution states:" << endl;
   cout << "solution:" << endl;
   for (size_t x = 0; x < data.shape(0)*2 - 1; ++x) {
      cout << left << setw(3) << setprecision(1) << "___";
   }
   cout << endl;
   for (size_t y = 0; y < data.shape(1)*2 - 1; ++y) {
      cout << "|";
      for (size_t x = 0; x < data.shape(0)*2 - 1; ++x) {
         if (x % 2 == 0 && y % 2 == 0) {
            data(x / 2, y / 2) = static_cast<float> (rand() % 10) *0.1;
            cout << left << setw(3) << setprecision(1) << " ";
         } else if (x % 2 == 0 && y % 2 == 1) {
            if (solution[cTHelper(x, y)]) {
               cout << left << setw(3) << setprecision(1) << "___";
            }
            else {
               cout << left << setw(3) << setprecision(1) << "   ";
            }
         } else if (x % 2 == 1 && y % 2 == 0) {
            if (solution[cTHelper(x, y)])
               cout << left << setw(3) << setprecision(1) << " | ";
            else
               cout << left << setw(3) << setprecision(1) << "   ";
         } else if (x % 2 == 1 && y % 2 == 1) {
            cout << left << setw(3) << setprecision(1) << " * ";
         }
      }
      cout << "|" << endl;
   }
   for (size_t x = 0; x < data.shape(0)*2 - 1; ++x) {
      cout << left << setw(3) << setprecision(1) << "___";
   }
   cout << endl;
}

// user defined Function Type
```

```
template<class T>
struct ClosednessFunctor {
public:
   typedef T value_type;

   template<class Iterator>
   inline const T operator()(Iterator begin)const {
      size_t sum = begin[0];
      sum += begin[1];
      sum += begin[2];
      sum += begin[3];
      if (sum != 2 && sum != 0) {
         return high;
      }
      return 0;
   };

   size_t dimension()const {
      return 4;
   };

   size_t shape(const size_t i)const {
      return 2;
   };

   size_t size()const {
      return 16;
   };
   T high;
};

int main(int argc, char** argv) {
   // model parameters
   const size_t gridSizeX = 5, gridSizeY = 5; //size of grid
   const float beta = 0.9; // bias to choose between under- and over-segmentation
   const float high = 10; // closedness-enforcing soft-constraint

   // size of the topological grid
   const size_t tGridSizeX = 2 * gridSizeX - 1, tGridSizeY = 2 * gridSizeY - 1;
   const size_t nrOfVariables = gridSizeY * (gridSizeX - 1) + gridSizeX * (gridSizeY - 1);
   const size_t dimT[] = {tGridSizeX, tGridSizeY};
   TopologicalCoordinateToIndex cTHelper(gridSizeX, gridSizeY);
   marray::Marray<float> data;
   randomData(gridSizeX, gridSizeY, data);

   cout << "interpixel_boundary_segmentation_with_closedness:" << endl;
   printData(data);

   // construct a graphical model with
   // - addition as the operation (template parameter Adder)
   // - the user defined function type ClosednessFunctor<float>
   // - gridSizeY * (gridSizeX - 1) + gridSizeX * (gridSizeY - 1) variables,
   // each having 2 many labels.
   typedef opengm::meta::TypeListGenerator<
      opengm::ExplicitFunction<float>,
      ClosednessFunctor<float>
   >::type FunctionTypeList;
   typedef opengm::GraphicalModel<float, opengm::Adder, FunctionTypeList,
      opengm::SimpleDiscreteSpace<> > Model;
   typedef Model::FunctionIdentifier FunctionIdentifier;
   Model gm(opengm::SimpleDiscreteSpace<>(nrOfVariables, 2));

   // for each boundary in the grid, i.e. for each variable
   // of the model, add one 1st order functions
   // and one 1st order factor
   {
      const size_t shape[] = {2};
```

```
    opengm::ExplicitFunction<float> f(shape, shape + 1);
    for (size_t yT = 0; yT < dimT[1]; ++yT) {
        for (size_t xT = 0; xT < dimT[0]; ++xT) {
            if ((xT % 2 + yT % 2) == 1) {
                float gradient = fabs(data(xT / 2, yT / 2) - data(xT / 2 + xT % 2, yT / 2 + yT % 2));
                f(0) = beta * gradient; // value for inactive boundary
                f(1) = (1.0 - beta) * (1.0 - gradient); // value for active boundary;
                FunctionIdentifier id = gm.addFunction(f);
                size_t vi[] = {cTHelper(xT, yT)};
                gm.addFactor(id, vi, vi + 1);
            }
        }
    }
}

// for each junction of four inter-pixel edges on the grid,
// one factor is added that connects the corresponding variable
// indices and refers to the ClosednessFunctor function
{
    // add one (!) 4th order ClosednessFunctor function
    ClosednessFunctor<float> f;
    f.high = high;
    FunctionIdentifier id = gm.addFunction(f);
    // add factors
    for (size_t y = 0; y < dimT[1]; ++y) {
        for (size_t x = 0; x < dimT[0]; ++x) {
            if (x % 2 + y % 2 == 2) {
                size_t vi[] = {
                    cTHelper(x + 1, y),
                    cTHelper(x - 1, y),
                    cTHelper(x, y + 1),
                    cTHelper(x, y - 1)
                };
                sort(vi, vi + 4);
                gm.addFactor(id, vi, vi + 4);
            }
        }
    }
}

// set up the optimizer (lazy flipper)
typedef opengm::LazyFlipper<Model, opengm::Minimizer> LazyFlipperType;
LazyFlipperType::VerboseVisitorType verboseVisitor;
size_t maxSubgraphSize = 5;
LazyFlipperType lazyflipper(gm, maxSubgraphSize);
cout << "start_inference:" << endl;

// obtain the (approximate) argmin
lazyflipper.infer(verboseVisitor);

// output the (approximate) argmin
vector<size_t> solution;
lazyflipper.arg(solution);
printSolution(data, solution);
}
```

Chapter 8

License

Bibliography

[1] B. Andres, J. H. Kappes, U. Koethe, and F. A. Hamprecht. The Lazy Flipper: MAP inference in higher-order graphical models by depth-limited exhaustive search. *ArXiv e-prints*, 2010.

[2] B. Andres, T. Koethe, U. Kroeger, and F. A. Hamprecht. Runtime-flexible multi-dimensional arrays and views for C++98 and C++0x. *ArXiv e-prints*, Aug. 2010.

[3] A. Barbu and S.-C. Zhu. Generalizing swendsen-wang to sampling arbitrary posterior probabilities. *TPAMI*, 27(8):1239–1253, 2005.

[4] M. Bergtholdt, J. H. Kappes, S. Schmidt, and C. Schnörr. A study of parts-based object class detection using complete graphs. *International Journal of Computer Vision*, 87(1-2):93–117, 2010.

[5] J. Besag. On the statisical analysis of dirty pictures. *Journal of the Royal Statistical Society B*, 48:259–302, 1986.

[6] E. Boros, P. L. Hammer, and X. Sun. Network flows and minimization of quadratic pseudo-boolean functions. Technical report, Technical Report RRR 17-1991, RUTCOR Research Report, 1991.

[7] E. Boros, P. L. Hammer, and G. Tavares. Preprocessing of unconstrained quadratic binary optimization. Technical report, Technical Report RRR 10-2006, RUTCOR Research Report, 2006.

[8] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *TPAMI*, 23(11):1222–1239, 2001.

[9] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *TPAMI*, 6:721–741, 1984.

[10] A. V. Goldberg, S. Hed, H. Kaplan, R. E. Tarjan, and R. F. Werneck. Maximum flows by incremental breadth-first search. In *Proceedings of the 19th European conference on Algorithms*, ESA'11, pages 457–468, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] P. Hammer, P. Hansen, and B. Simeone. Roof duality, complementation and persistency in quadratic 0-1 optimization. *Math. Programming*, 28:121–155, 1984.

[12] HDF5 Group. HDF5.
urlhttp://www.hdfgroup.org/HDF5/.

[13] S. Hed. IBFS.
urlhttp://www.cs.tau.ac.il/ sagihed/ibfs/.

[14] C. Helmberg. The ConicBundle Library for Convex Optimization v0.3.11.
urlhttp://www-user.tu-chemnitz.de/ helmberg/ConicBundle/, 2012.

[15] IBM. IBM ILOG CPLEX Optimizer.
urlhttp://www-01.ibm.com/software/integration/optimization/cplex-optimizer/, 2012.

[16] Jeremy G. Siek et.al.. Boost C++ Libraries v1.47.0.
urlhttp://www.boost.org/, 2011.

[17] K. Jung, P. Kohli, and D. Shah. Local rules for global map: When do they work ? In *NIPS*, 2009.

[18] J. H. Kappes. *Inference on Highly-Connected Discrete Graphical Models with Applications to Visual Object Recognition*. doctoral thesis, Ruprecht-Karls-Universität Heidelberg, Faculty of Mathematics and Computer Sciences, Heidelberg, Germany, 2011.

[19] J. H. Kappes, B. Savchynskyy, and C. Schnörr. A bundle approach to efficient MAP-inference by Lagrangian relaxation. In *CVPR 2012*, 2012. in press.

[20] P. Kohli, M. P. Kumar, and P. H. S. Torr. P3 & beyond: Solving energies with higher order cliques. In *CVPR*, 2007.

[21] V. Kolmogorov. Maxflow v3.02.
urlhttp://pub.ist.ac.at/ vnk/software.html.

[22] V. Kolmogorov. QPBO v1.3.
urlhttp://pub.ist.ac.at/ vnk/software.html.

[23] V. Kolmogorov. TRW-S v1.3.
urlhttp://research.microsoft.com/en-us/downloads/dad6c31e-2c04-471f-b724-ded18bf70fe3/.

[24] V. Kolmogorov. Convergent tree-reweighted message passing for energy minimization. *TPAMI*, 28(10):1568–1583, 2006.

[25] N. Komodakis, N. Paragios, and G. Tziritas. MRF energy minimization and beyond via dual decomposition. *TPAMI*, 33(3):531–552, 2011.

[26] F. R. Kschischang, B. J. Frey, and H. Loeliger. Factor graphs and the sum-product algorithm. *Transactions on Information Theory*, 47:498–519, 2001.

[27] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[28] J. M. Mooij. libDAI v0.3.0.
urlhttp://cs.ru.nl/ jorism/libDAI/.

[29] J. M. Mooij. libDAI: A free and open source C++ library for discrete approximate inference in graphical models. *JMLR*, 11:2169–2173, Aug. 2010.

[30] K. Murphy, Y. Weiss, and M. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *UAI*, 1999.

[31] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, San Francisco, CA, USA, 1988.

[32] C. Rother, V. Kolmogorov, V. S. Lempitsky, and M. Szummer. Optimizing binary mrfs via extended roof duality. In *CVPR*, 2007.

[33] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen, and C. Rother. MRF-LIB 2.1.
urlhttp://vision.middlebury.edu/MRF/code/.

[34] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen, and C. Rother. A comparative study of energy minimization methods for markov random fields with smoothness-based priors. *TPMAI*, 30(6):1068–1080, 2008.

[35] M. J. Wainwright and M. I. Jordan. *Graphical Models, Exponential Families, and Variational Inference*. Now Publishers Inc., Hanover, MA, USA, 2008.