Submodular library version 1.0

Petter Strandmark

October 3, 2011

Contents

1 Included in the package

The library contains the source code for the paper "Generalized Roof Duality for Pseudo-Boolean Optimization". Included is also some code for heuristics, which does not use linear programming and is therefore much faster. These techniques have not been published so far and should be considered experimental.

This library handles polynomials of degrees 3 and 4. The degree-2 case corresponds to regular roof duality.

2 Compatibility

This software can be compiled with Microsoft Visual C++ 2010 or GCC 4.5 or later. Using earlier compilers is not possible, due to the use of C++0x features. The following compilers have been tested:

- Microsoft Visual C++ 2010 (32-bit and 64-bit)
- g++ (GCC) 4.5.0 (32-bit Cygwin)
- g++-4.5 (Ubuntu/Linaro 4.5.1-7ubuntu2) 4.5.1 (64-bit)
- g++ (Ubuntu/Linaro 4.5.2-8ubuntu4) 4.5.2 (64-bit)
- g++ (Ubuntu/Linaro 4.6.1-9ubuntu3) 4.6.1 (64-bit)

Under g++, the program compiles with "-Wall -Werr -pedantic-errors", so hopefully it should be reasonably portable.

3 Third-party libraries

Due to the use of linear programming and maximum flow, some third-party libraries cannot be avoided:

- Clp. See the next section about compilation on how to check out and compile the source code. Note that **Clp is not required** if you omit `PseudoBoolean_create_g.cpp` from compilation, i.e. only use heuristics. Then the only requirements are the packages below.

The following third-party libraries, well-known to the computer vision community, are also required:

- HOCR1.02 (not an eariler version)
- maxflow-v3.01
- QPBO-v1.3

The can be downloaded to the correct place automatically by executing the script `download_thirdparty.sh` from the `thirdparty` folder. Alternatively, the subfolders in the `thirdparty` folder contain the links to the appropriate download pages.

4 Compilation

4.1 Microsoft Visual C++

Project files for Microsoft Visual Studio 2010 are included. Compiling the software in 64-bit mode is recommended.

4.2 gcc

Clp has to be downloaded and compiled. This should not be very difficult. The following instructions are also available at `https://projects.coin-or.org/Clp` :

1. svn co https://projects.coin-or.org/svn/Clp/stable/1.12 coin-Clp
2. cd coin-Clp
3. ./configure -C
4. make
5. make test
6. make install

Then the Makefile has to be edited so that CLPINCLUDEDIR and CLPLIBDIR point to the correct location. After this, the software can be compiled with

1. make

To run the program, the environment variable LD_LIBRARY_PATH should be edited to include coin-Clp/lib or you can copy its contents to a standard library path such as /usr/local/lib if you have root access. Under Windows/Cygwin I had to make the following two changes in order to compile Clp:

- Get Cygwin make v.3.81; see `https://projects.coin-or.org/BuildTools/wiki/current-issues`
- Add two lines to `CoinParam.hpp`:

```
#include <cstdio>
using std::FILE;
```

5 Usage

The library consists of two main classes:

- PseudoBoolean – this class represents a pseudo-boolean function, corresponding to $f(\boldsymbol{x})$ in the paper.
- SymmetricPseudoBoolean – represents a symmetric pseudo-boolean function, corresponding to $g(\boldsymbol{x}, \boldsymbol{y})$ in the paper.

5.1 PseudoBoolean

The PseudoBoolean class is usually created explicity by adding the monomials one at a time:

- PseudoBoolean::add_monomial(**int** i, real a);
- PseudoBoolean::add_monomial(**int** i, **int** j, real a);
- PseudoBoolean::add_monomial(**int** i, **int** j, **int** k, real a);
- PseudoBoolean::add_monomial(**int** i, **int** j, **int** k, **int** l, real a);

Alternatively, the function can be created as a sum of cliques:

- PseudoBoolean::add_clique(**int** i, real E0, real E1);
- PseudoBoolean::add_clique(**int** i, **int** j, real E00 ... real E11);
- PseudoBoolean::add_clique(**int** i, **int** j, **int** k, **const** vector<real>& E);
- PseudoBoolean::add_clique(**int** i, **int** j, **int** k, real E000 ... real E111);
- PseudoBoolean::add_clique(**int** i, **int** j, **int** k, **int** l, **const** vector<real>\& E);
- PseudoBoolean::add_clique(**int** i, **int** j, **int** k, **int** l, real E0000...real E1111);

5.2 SymmetricPseudoBoolean

The SymmetricPseudoBoolean class is created indirectly from a PseudoBoolean, either by solving a linear program or by heuristic methods. The relation $g(\boldsymbol{x}, \bar{\boldsymbol{x}}) = f(\boldsymbol{x})$ will always hold.

5.3 Example usage

The simplest usage case is something like the following:

```
using namespace Petter;
PseudoBoolean f;

// Create f by calling the add_clique and add_term functions
f.add_clique( ... );
// ...
f.add_clique( ... );


vector<label> x(n); // Holds the solution x
int labeled; // Will hold the number of labeled variables
bool use_heuristic = false;
f.minimize(x, labeled, use_heuristic);
cout << "Optimal relaxation labeled " << labeled << " variables\n";

use_heuristic = true;
f.minimize(x, labeled, use_heuristic);
cout << "Heuristic relaxation labeled " << labeled << " variables\n";

f.minimize_reduction(x, labeled);
cout << "HOCR labeled " << labeled << " variables\n";
```

The solution will end up in the vector x, which takes the values

$$\mathrm{x}_n = \begin{cases} 0, & \text{if } x_n = 0 \text{ by persistency, } (x,y) = (0,1) \\ 1, & \text{if } x_n = 1 \text{ by persistency, } (x,y) = (1,0) \\ -1, & \text{if no persistency was found, } (x,y) = (0,0) \iff \text{unlabeled.} \end{cases} \tag{1}$$

This is in accordance with the popular "QPBO" software written by Vladimir Kolmogorov.

The function PseudoBoolean::minimize performs all steps required for the minimization (creating and minimizing g, reducing f etc.). Its source code is listed below:

```
real PseudoBoolean::minimize(vector<label>& x, int& labeled, bool heuristic)
{
        bool should_continue;
        real bound;
        labeled = 0;
```

```
        int n = int( x.size() );

        do {
                // Create symmetric relaxation
                SymmetricPseudoBoolean spb;
                if (heuristic) {
                        spb.create_heuristic(*this);
                }
                else {
                        spb.create_lp(*this);
                }

                // Minimize relaxation
                int new_labeled = 0;
                bound = spb.minimize(x, new_labeled);

                // If we have more persistencies, continue
                should_continue = new_labeled > labeled;
                labeled = new_labeled;
                if (labeled == n) {
                        //Nothing more to do
                        should_continue = false;
                }
                // Reduce this function
                reduce(x);

        } while (should_continue);

        return bound;
}
```

5.4 Example program

The makefile and the project will compile a program called `submodular` which demonstrates various functionality. Its options are:

`-n` n	Generates a random polynomial with n variables.
`-m` m	The random polynomial will have degree m.
`-nterms` k	The random polynomial will consist of k cliques.
`-example`	Uses the examples from the paper (both for $m = 3$ and $m = 4$)
`-file` *file*	Reads polynomial from file. Example files are located in the `tests` folder.
`-sat` *file*	Reads a satisfiability problem (3SAT) from file.
`-heuristic`	Uses heuristics as well.