Listing 1: Brute force

```
/*
 * lp_brute_force.hpp
 * Solving linear programming problem by brute force.
 * Vladimir Rutsky <altsysrq@gmail.com>
 * 22.05.2009
 */

#ifndef NUMERIC_LP_BRUTE_FORCE_HPP
#define NUMERIC_LP_BRUTE_FORCE_HPP

#include "numeric_common.hpp"

#include <algorithm>
#include <vector>

#include <boost/assert.hpp>
#include <boost/concept/assert.hpp>
#include <boost/concept_check.hpp>

#include "linear_problem.hpp"
#include "linear_problem_algs.hpp"
#include "simplex_alg.hpp"
#include "linear_system.hpp"
#include "matrix_ops.hpp"

namespace numeric
{
namespace linear_problem
{
  template< class E, class CLPTraits >
  simplex::simplex_result_type
    solve_by_brute_force( ICommonLinearProblem<CLPTraits> const &commonLP,
                          vector_expression<E> &result )
  {
    typedef CLPTraits                                clp_traits_type;
    typedef typename CLPTraits::scalar_type          scalar_type;
    typedef vector<scalar_type>                      vector_type;
    typedef zero_vector<scalar_type>                 zero_vector_type;

    typedef common_linear_problem    <scalar_type, clp_traits_type> common_linear_problem_type;
    typedef canonical_linear_problem<scalar_type, clp_traits_type> canonical_linear_problem_type;
    typedef typename converter_template_type <scalar_type >::type    converter_type;

    typedef typename vector_type::size_type          size_type;
    typedef std::vector<size_type>                   indexes_container_type;

    // Converting linear problem to canonical form.
    canonical_linear_problem_type canonicalLP;
    converter_type conv = to_canonical(commonLP, canonicalLP);
    BOOST_ASSERT(assert_valid(canonicalLP));

    // Removing linear dependent constraints.
    canonical_linear_problem_type liCanonicalLP;
    if (!remove_dependent_constraints(canonicalLP, liCanonicalLP))
    {
      // Constraints are incosistent.
      return simplex::srt_none;
    }
    BOOST_ASSERT(assert_valid(liCanonicalLP));

    // Checking is dual problem have consistent constraints.
    common_linear_problem_type dualLP;
    construct_dual(commonLP, dualLP);
    BOOST_ASSERT(assert_valid(dualLP));

    canonical_linear_problem_type dualCanonicalLP;
    to_canonical(dualLP, dualCanonicalLP);
    BOOST_ASSERT(assert_valid(dualCanonicalLP));

    canonical_linear_problem_type liDualCanonicalLP;
    if (!remove_dependent_constraints(dualCanonicalLP, liDualCanonicalLP))
    {
      // Constraints of dual problem are incosistent, so in direct problem goal function don't have lower bound.
      return simplex::srt_not_limited;
    }
    BOOST_ASSERT(assert_valid(liDualCanonicalLP));

    {
      // Now we now that direct problem has exact solution.
      // Working with `liCanonicalLP' only.

      size_t const m = constraints_count(liCanonicalLP), n = variables_count(liCanonicalLP);

      // TODO
      BOOST_ASSERT(n > 0);
      BOOST_ASSERT(m > 0);

      // Iterating through all basic vectors and selecting one that minimizes goal function.
      boost::optional<std::pair<scalar_type, vector_type> > minVec;
      size_t nFoundedBasicVecs(0); // debug

      size_t combinationsNum(0); // debug
      indexes_container_type idxs;
      combination::first_combination<size_type>(std::back_inserter(idxs), m);

      do
      {
        ++combinationsNum;

        BOOST_ASSERT(std::adjacent_find(idxs.begin(), idxs.end(), std::greater<size_type>()) == idxs.end());
        BOOST_ASSERT(idxs.size() == m);

        bool const isLI = is_linear_independent(
            matrix_columns_begin(submatrixi(liCanonicalLP.A(), size_t(0), m, idxs.begin(), idxs.end())),
            matrix_columns_end  (submatrixi(liCanonicalLP.A(), size_t(0), m, idxs.begin(), idxs.end())));

        if (isLI)
        {
          // Calculating linear system solution.
          vector_type basicSubvector;
          BOOST_VERIFY(linear_system::solve(
              submatrixi(liCanonicalLP.A(), size_t(0), m, idxs.begin(), idxs.end()),
              liCanonicalLP.b(),
              basicSubvector));

          // Calculating vector corresponding to solition.
          vector_type basicVector = zero_vector_type(n);
          BOOST_ASSERT(idxs.size() == m);
          for (size_t r = 0; r < m; ++r)
            basicVector[idxs[r]] = basicSubvector[r];

          if (std::find_if(basicVector.begin(), basicVector.end(), boost::bind<bool>(std::less<scalar_type>(), _1, 0.)) == basicVector.end())
          {
            // Found actual basic vector.
            ++nFoundedBasicVecs;

            BOOST_ASSERT(simplex::assert_basic_vector(liCanonicalLP.A(), liCanonicalLP.b(), basicVector));

            // Saving minimum basic vector between old and new one.
            BOOST_ASSERT(liCanonicalLP.c().size() == basicVector.size());
            scalar_type const goalFuncVal = std::inner_product(
                liCanonicalLP.c().begin(), liCanonicalLP.c().end(), basicVector.begin(), scalar_type(0));

            if (!minVec || minVec->first > goalFuncVal)
              minVec = std::make_pair(goalFuncVal, basicVector);
          }
        }
      } while (combination::next_combination(idxs.begin(), n, m));

      BOOST_ASSERT(minVec);

      result() = conv(minVec->second);

      return simplex::srt_min_found;
    }
  }

  template< class CLPTraits >
  bool is_brute_force_solving_correct( ICommonLinearProblem<CLPTraits> const &commonLP )
  {
    typedef CLPTraits                              clp_traits_type;
    typedef typename CLPTraits::scalar_type        scalar_type;
    typedef vector<scalar_type>                    vector_type;

    typedef common_linear_problem<scalar_type, clp_traits_type> common_linear_problem_type;

    // Solving direct linear problem.
    vector_type directResultVec;
    simplex::simplex_result_type const directSimplexResult = solve_by_brute_force(commonLP, directResultVec);
    BOOST_ASSERT(directSimplexResult != simplex::srt_loop); // TODO: Not implemented bahavior.

    // Constructing dual linear problem.
    common_linear_problem_type dualLP;
    construct_dual(commonLP, dualLP);

    // Solving dual linear problem.
    vector_type dualResultVec;
    simplex::simplex_result_type const dualSimplexResult = solve_by_brute_force(dualLP, dualResultVec);
    BOOST_ASSERT(dualSimplexResult != simplex::srt_loop); // TODO: Not implemented bahavior.

    if (!is_lp_solution_correct(commonLP,
                                directSimplexResult, directResultVec,
                                dualSimplexResult, dualResultVec))
    {
      return false;
    }

    return true;
  }
} // End of namespace 'linear_problem'.
} // End of namespace 'numeric'.

#endif // NUMERIC_LP_BRUTE_FORCE_HPP
```