Listing 1: Method of potentials

```
/*
 * potentials_alg.hpp
 * Method of potentials for solving transportation problem.
 * Vladimir Rutsky <altsysrq@gmail.com>
 * 25.05.2009
 */

#ifndef NUMERIC_POTENTIALS_ALG_HPP
#define NUMERIC_POTENTIALS_ALG_HPP

#include <algorithm>
#include <vector>
#include <map>
#include <utility>
#include <queue>
#include <list>

#include "numeric_common.hpp"

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

#include <boost/bind.hpp>
#include <boost/shared_ptr.hpp>
#include <boost/optional.hpp>
#include <boost/lambda/lambda.hpp>
#include <boost/utility.hpp>

#include "transportation_problem.hpp"
#include "matrix_ops.hpp"

namespace numeric
{
namespace lp_potentials
{
  template< class V1, class V2, class M >
  bool is_tp_valid( vector_expression<V1> const &a, vector_expression<V2> const &b,
                    matrix_expression<M>  const &C )
  {
    typedef typename V1::value_type scalar_type; // TODO

    if (!(C().size1() > 0 && C().size2() > 0 && C().size1() == a().size() && C().size2()
        == b().size()))
    {
      // Data sizes is not correspond.
      return false;
    }

    if (!(
        std::find_if(a().begin(), a().end(),
                     boost::bind<bool>(std::less_equal<scalar_type>(), _1, scalar_type())
                         ) == a().end() &&
        std::find_if(b().begin(), b().end(),
                     boost::bind<bool>(std::less_equal<scalar_type>(), _1, scalar_type())
                         ) == b().end() &&
        std::find_if(C().data().begin(), C().data().end(),
                     boost::bind<bool>(std::less<scalar_type>(), _1, scalar_type())) == C
                         ().data().end()))
    {
      // One of data values is less than zero, which is incorrect.
      return false;
    }

    return true;
  }

  template< class V1, class V2, class M >
  bool assert_tp_valid( vector_expression<V1> const &a, vector_expression<V2> const &b,
                        matrix_expression<M>  const &C )
```

```
67  {
68    typedef typename V1::value_type scalar_type; // TODO
69
70    // Asserting sizes.
71    ASSERT_GT(a().size(), 0);             // a().size() > 0
72    ASSERT_GT(b().size(), 0);             // b().size() > 0
73    ASSERT_GT(C().size1(), 0);            // C().size1() > 0
74    ASSERT_GT(C().size2(), 0);            // C().size2() > 0
75    ASSERT_EQ(C().size1(), a().size()); // C().size1() == a().size()
76    ASSERT_EQ(C().size2(), b().size()); // C().size2() == b().size()
77
78    ASSERT(std::find_if(a().begin(), a().end(),
79                        boost::bind<bool>(std::less_equal<scalar_type>(), _1, scalar_type
                            ())) == a().end());
80    ASSERT(std::find_if(b().begin(), b().end(),
81                        boost::bind<bool>(std::less_equal<scalar_type>(), _1, scalar_type
                            ())) == b().end());
82    ASSERT(std::find_if(C().data().begin(), C().data().end(),
83                        boost::bind<bool>(std::less<scalar_type>(), _1, scalar_type()))
                            == C().data().end());
84
85    return is_tp_valid(a, b, C);
86  }
87
88  template< class V1, class V2, class M >
89  bool is_tp_closed( vector_expression<V1> const &a, vector_expression<V2> const &b,
90                     matrix_expression<M>  const &C )
91  {
92    typedef typename V1::value_type scalar_type; // TODO
93
94    BOOST_ASSERT(assert_tp_valid(a, b, C));
95
96    if (std::accumulate(a().begin(), a().end(), scalar_type()) ==
97        std::accumulate(b().begin(), b().end(), scalar_type()))
98    {
99      // Sum of supplies equal to sum of demand, problem is closed.
100     return true;
101   }
102   else
103   {
104     // Problem is unclosed.
105     return false;
106   }
107 }
108
109 template< class V1, class V2, class M >
110 bool is_plan( vector_expression<V1> const &a, vector_expression<V2> const &b,
111               matrix_expression<M> const &X )
112 {
113   typedef typename V1::value_type scalar_type;
114
115   size_t const m = a().size(), n = b().size();
116
117   ASSERT_EQ(X().size1(), m);
118   ASSERT_EQ(X().size2(), n);
119
120   for (size_t r = 0; r < m; ++r)
121     if (!eq(std::accumulate(row   (X(), r).begin(), row   (X(), r).end(), scalar_type()
          ), a()(r)))
122       return false;
123
124   for (size_t c = 0; c < n; ++c)
125     if (!eq(std::accumulate(column(X(), c).begin(), column(X(), c).end(), scalar_type()
          ), b()(c)))
126       return false;
127
128   return true;
129 }
130
131 template< class V1, class V2, class M >
132 bool assert_plan( vector_expression<V1> const &a, vector_expression<V2> const &b,
```

```
133                     matrix_expression<M> const &X )
134   {
135     typedef typename V1::value_type scalar_type;
136
137     size_t const m = a().size(), n = b().size();
138
139     ASSERT_EQ(X().size1(), m);
140     ASSERT_EQ(X().size2(), n);
141
142     for (size_t r = 0; r < m; ++r)
143     {
144       ASSERT_FUZZY_EQ(std::accumulate(row    (X(), r).begin(), row    (X(), r).end(),
              scalar_type()), a()(r));
145     }
146
147     for (size_t c = 0; c < n; ++c)
148     {
149       ASSERT_FUZZY_EQ(std::accumulate(column(X(), c).begin(), column(X(), c).end(),
              scalar_type()), b()(c));
150     }
151
152     return is_plan(a, b, X);
153   }
154
155   // TODO: Use 'details' namespace.
156   namespace
157   {
158     struct cell_type
159     {
160       cell_type()
161       {}
162
163       cell_type( cell_type const &other )
164         : r   (other.r)
165         , c   (other.c)
166         , mark(other.mark)
167       {}
168
169       cell_type( size_t newR, size_t newC )
170         : r(newR)
171         , c(newC)
172       {}
173
174       // Cell position in table.
175       size_t r, c;
176
177       // Cell mark, for DFS.
178       bool mark;
179     };
180
181     typedef boost::shared_ptr<cell_type>                          cell_ptr_type;
182     typedef std::map<size_t, cell_ptr_type>                       cells_map_type;
183     typedef std::vector<cells_map_type>                           cells_maps_vector_type;
184
185     typedef std::map<std::pair<size_t, size_t>, cell_ptr_type> all_cells_map_type;
186
187     template< class V, class M >
188     void build_start_plan( vector_expression<V> const &aVec, vector_expression<V> const &
            bVec,
189                            matrix_expression<M> const &C,
190                            matrix_expression<M>       &x,
191                            all_cells_map_type         &planCells )
192     {
193       typedef typename V::value_type     scalar_type; // TODO
194       typedef vector<scalar_type>        vector_type;
195       typedef matrix<scalar_type>        matrix_type;
196       typedef zero_matrix<scalar_type> zero_matrix_type;
197
198       vector_type a = aVec(), b = bVec();
199
200       ASSERT(assert_tp_valid(a, b, C));
```

```
201         ASSERT(is_tp_closed(a, b, C));
202
203         size_t const m = C().size1(), n = C().size2();
204
205         // Initializing list of unprocessed columns indexes.
206         typedef std::list<size_t> unprocessed_cols_list_type;
207         unprocessed_cols_list_type unprocessedCols;
208         basic_range<size_t, long> N(0, n);
209         unprocessedCols.assign(N.begin(), N.end());
210
211         // Resetting result data().
212         planCells.clear();
213         x() = zero_matrix_type(m, n);
214
215         // Building start plan.
216         for (size_t r = 0; r < m; ++r)
217         {
218           scalar_type &supply = a(r);
219
220           while (true)
221           {
222             bool const fake = eq_zero(supply);
223
224             ASSERT(!unprocessedCols.empty());
225
226             // Locating column with lowest transfer cost from current row.
227             boost::optional<size_t> minElemIdx;
228             for (unprocessed_cols_list_type::const_iterator it = unprocessedCols.begin();
                      it != unprocessedCols.end(); ++it)
229               if (!minElemIdx || C()(r, *it) < C()(r, minElemIdx.get()))
230                 minElemIdx = *it;
231             size_t const minc = *minElemIdx;
232
233             scalar_type &demand = b(minc);
234
235             if (supply < demand)
236             {
237               // Left supply can't satisfy current demand.
238
239               // Decreasing demand by left supply value.
240               scalar_type const transfer = supply;
241               ASSERT_EQ(x()(r, minc), scalar_type());
242               x()(r, minc) = transfer;
243               demand      -= transfer;
244               supply      -= transfer; supply = 0;
245
246               // Adding current cell into plan.
247               VERIFY(planCells.insert(std::make_pair(std::make_pair(r, minc), cell_ptr_type
                          (new cell_type(r, minc)))).second);
248
249               // Interrupting row processing.
250               break;
251             }
252             else if (supply > demand)
253             {
254               // Left supply is greater that current demand.
255
256               // Satisfying demand.
257               scalar_type const transfer = demand;
258               ASSERT_EQ(x()(r, minc), scalar_type());
259               x()(r, minc) = transfer;
260               demand      -= transfer; demand = 0;
261               supply      -= transfer;
262
263               // Adding current cell into plan.
264               VERIFY(planCells.insert(std::make_pair(std::make_pair(r, minc), cell_ptr_type
                          (new cell_type(r, minc)))).second);
265
266               // Removing current demand from unprocessed list.
267               unprocessed_cols_list_type::iterator it = std::remove(unprocessedCols.begin()
                          , unprocessedCols.end(), minc);
```

```
268             ASSERT(it != unprocessedCols.end());
269             ASSERT(boost::next(it, 1) == unprocessedCols.end());
270             unprocessedCols.erase(it, unprocessedCols.end());
271 
272             // Continuing on current row.
273           }
274           else // supply == demand
275           {
276             // Left supply exactly satisfies demand.
277             // Satisfying demand
278             scalar_type const transfer = demand;
279             ASSERT_EQ(x()(r, minc), scalar_type());
280             x()(r, minc) = transfer;
281             demand      -= transfer; demand = 0;
282             supply      -= transfer; supply = 0;
283 
284             // Adding current cell into plan
285             VERIFY(planCells.insert(std::make_pair(std::make_pair(r, minc), cell_ptr_type
                    (new cell_type(r, minc)))).second);
286 
287             // Removing current demand from unprocessed list.
288             unprocessed_cols_list_type::iterator it = std::remove(unprocessedCols.begin()
                    , unprocessedCols.end(), minc);
289             ASSERT(it != unprocessedCols.end());
290             ASSERT(boost::next(it, 1) == unprocessedCols.end());
291             unprocessedCols.erase(it, unprocessedCols.end());
292 
293             // Continuing on current row (for adding fake plan cell).
294             if (r == m - 1)
295             {
296               // But no fake elements at last row.
297               break;
298             }
299           }
300 
301           if (fake)
302           {
303             // Not more than one fake element per row.
304             break;
305           }
306         }
307       }
308 
309       // Asserting that no supplies or demands are left.
310       ASSERT_EQ(norm_2(a), 0);
311       ASSERT_EQ(norm_2(b), 0);
312 
313       // Asserting that number of plan points is exactly m + n - 1.
314       ASSERT_EQ(planCells.size(), m + n - 1);
315 
316       // Asserting that founded x is a plan.
317       ASSERT(assert_plan(aVec, bVec, x));
318     }
319 
320     template< class M, class V >
321     void calculate_potentials_coefs( cells_maps_vector_type const &rows,
            cells_maps_vector_type const &cols,
322                                      matrix_expression<M> const &C,
323                                      vector_expression<V> &uVec, vector_expression<V> &
                                         vVec )
324     {
325       typedef typename M::value_type scalar_type; // TODO
326 
327       size_t const m = C().size1(), n = C().size2();
328 
329       ASSERT_EQ(rows.size(), m); // rows.size() == m
330       ASSERT_EQ(cols.size(), n); // cols.size() == n
331 
332       std::vector<boost::optional<scalar_type> > u(m), v(n);
333       std::queue<size_t> rowsQueue, colsQueue;
334 
```

```
335         // Setting u(0) to zero and adding first row to rows queue.
336         u[0] = 0;
337         rowsQueue.push(0);
338
339         while (!rowsQueue.empty() || !colsQueue.empty())
340         {
341           if (!rowsQueue.empty())
342           {
343             // Poping row index from rows queue.
344             size_t const r = rowsQueue.front();
345             rowsQueue.pop();
346
347             for (cells_map_type::const_iterator cellPtrIt = rows[r].begin(); cellPtrIt !=
                    rows[r].end(); ++cellPtrIt)
348             {
349               cell_type const &cell = *(cellPtrIt->second);
350               size_t const c = cell.c;
351               ASSERT_LT(c, n);       // c < n
352               ASSERT_EQ(cell.r, r); // cell.r == r
353
354               if (!v[c])
355               {
356                 // For each unprocessed column calculating potential coefficient v[c]:
357                 //   u[r] + v[c] = C[r, c]   <=>
358                 //   v[c] = C[r, c] - u[r]
359                 v[c] = C()(r, c) - u[r].get();
360
361                 // And adding processed columns to columns queue.
362                 colsQueue.push(c);
363               }
364               else
365               {
366                 // Asserting that there is no conflicts.
367                 ASSERT_EQ(v[c].get(), C()(r, c) - u[r].get());
368               }
369             }
370           }
371
372           if (!colsQueue.empty())
373           {
374             // Poping row index from columns queue.
375             size_t const c = colsQueue.front();
376             colsQueue.pop();
377
378             for (cells_map_type::const_iterator cellPtrIt = cols[c].begin(); cellPtrIt !=
                    cols[c].end(); ++cellPtrIt)
379             {
380               cell_type const &cell = *(cellPtrIt->second);
381               size_t const r = cell.r;
382               ASSERT_LT(r, m);       // r < m
383               ASSERT_EQ(cell.c, c); // cell.c == c
384
385               if (!u[r])
386               {
387                 // For each unprocessed column calculating potential coefficient v[c]:
388                 //   u[r] + v[c] = C[r, c]   <=>
389                 //   u[r] = C[r, c] - v[c]
390                 u[r] = C()(r, c) - v[c].get();
391
392                 // And adding processed rows to rows queue.
393                 rowsQueue.push(r);
394               }
395               else
396               {
397                 // Asserting that there is no conflicts.
398                 ASSERT_EQ(u[r], C()(r, c) - v[c].get());
399               }
400             }
401           }
402         }
403
```

```
404         uVec().resize(m);
405         std::transform(u.begin(), u.end(), uVec().begin(), boost::lambda::ret<scalar_type
                >(*boost::lambda::_1));
406         vVec().resize(n);
407         std::transform(v.begin(), v.end(), vVec().begin(), boost::lambda::ret<scalar_type
                >(*boost::lambda::_1));
408       }
409
410       template< class M, class V >
411       void calculate_potentials( vector_expression<V> const &u, vector_expression<V> const
              &v,
412                                  matrix_expression<M> const &C,
413                                  matrix_expression<M>       &P )
414       {
415         typedef typename V::value_type  scalar_type; // TODO
416         typedef vector<scalar_type>       vector_type;
417         typedef zero_vector<scalar_type> zero_vector_type;
418         typedef matrix<scalar_type>       matrix_type;
419         typedef zero_matrix<scalar_type> zero_matrix_type;
420
421         size_t const m = u().size(), n = v().size();
422
423         // Asserting sizes.
424         ASSERT_GT(m, 0);             // m > 0
425         ASSERT_GT(n, 0);             // n > 0
426         ASSERT_GT(C().size1(), 0); // C().size1() > 0
427         ASSERT_GT(C().size2(), 0); // C().size2() > 0
428         ASSERT_EQ(C().size1(), m); // C().size1() == m
429         ASSERT_EQ(C().size2(), n); // C().size2() == n
430
431         P().resize(m, n);
432
433         for (size_t r = 0; r < m; ++r)
434           for (size_t c = 0; c < n; ++c)
435           {
436             P()(r, c) = u()(r) + v()(c) - C()(r, c);
437           }
438       }
439
440       template< class IdxsOutputIterator >
441       bool find_loop( cells_maps_vector_type const &rows, cells_maps_vector_type const &
              cols,
442                       size_t r, size_t c, size_t const goalR, size_t const goalC,
443                       IdxsOutputIterator idxsOut, bool horizontalSearch, size_t depth )
444       {
445         // TODO: Assert sizes.
446
447         // 'depth' equals to number of cells in current 'loop' (current cell is counted).
448         // 'horizontalSearch' is true if from current cell must be searched cells only on
                current cell row.
449
450         // Adding current point to building loop end on visit.
451         if (!(r == goalR && c == goalC))
452         {
453           // Not at first call.
454
455           ASSERT(rows[r].find(c) != rows[r].end());
456           cell_type &curCell = *rows[r].find(c)->second;
457
458           curCell.mark = true;
459         }
460
461         if (depth >= 3)
462         {
463           // Starting from depth 3 looking for ability to close loop.
464           if ((horizontalSearch && r == goalR) || (!horizontalSearch && c == goalC))
465           {
466             // Goal cell is seen from current cell. Closing loop.
467             *idxsOut++ = std::make_pair(r, c);
468             return true;
469           }
```

```
470         }
471
472         if (horizontalSearch)
473         {
474           // Searching for next cell to append to loop in the row of loop end.
475           for (cells_map_type::const_iterator cellPtrIt = rows[r].begin(); cellPtrIt !=
                  rows[r].end(); ++cellPtrIt)
476           {
477             cell_type const &cell = *(cellPtrIt->second);
478
479             if (r == cell.r && c == cell.c)
480             {
481               // Omitting current cell.
482               continue;
483             }
484             ASSERT(!(cell.r == goalR && cell.c == goalC));
485
486             if (!cell.mark)
487             {
488               // Found cell not in current loop, trying to append it to current loop.
489               if (find_loop(rows, cols, cell.r, cell.c, goalR, goalC, idxsOut, !
                      horizontalSearch, depth + 1))
490               {
491                 // Loop was found, outputting it.
492                 *idxsOut++ = std::make_pair(r, c);
493                 return true;
494               }
495             }
496           }
497         }
498         else
499         {
500           // Searching for next cell to append to loop in the column of loop end.
501           for (cells_map_type::const_iterator cellPtrIt = cols[c].begin(); cellPtrIt !=
                  cols[c].end(); ++cellPtrIt)
502           {
503             cell_type const &cell = *(cellPtrIt->second);
504
505             if (r == cell.r && c == cell.c)
506             {
507               // Omitting current cell.
508               continue;
509             }
510             ASSERT(!(cell.r == goalR && cell.c == goalC));
511
512             if (!cell.mark)
513             {
514               // Found cell not in current loop, trying to append it to current loop.
515               if (find_loop(rows, cols, cell.r, cell.c, goalR, goalC, idxsOut, !
                      horizontalSearch, depth + 1))
516               {
517                 // Loop was found, outputting it.
518                 *idxsOut++ = std::make_pair(r, c);
519                 return true;
520               }
521             }
522           }
523         }
524
525         // Nothing useful found, removing current point from loop end.
526         if (!(r == goalR && c == goalC))
527         {
528           // Not at first call.
529
530           ASSERT(rows[r].find(c) != rows[r].end());
531           cell_type &curCell = *rows[r].find(c)->second;
532
533           curCell.mark = false;
534         }
535
536         return false;
```

```
537     }
538
539     template< class IdxsOutputIterator >
540     void find_loop( cells_maps_vector_type const &rows, cells_maps_vector_type const &
            cols,
541                     size_t const goalR, size_t const goalC,
542                     IdxsOutputIterator idxsOut )
543     {
544       // TODO: Assert sizes.
545       size_t const m = rows.size(), n = cols.size();
546
547       // Resetting cells marks.
548       for (size_t r = 0; r < m; ++r)
549         for (cells_map_type::const_iterator cellPtrIt = rows[r].begin(); cellPtrIt !=
              rows[r].end(); ++cellPtrIt)
550         {
551           cell_type &cell = *(cellPtrIt->second);
552           size_t const c = cell.c;
553           ASSERT_LT(c, n);       // c <= n
554           ASSERT_EQ(cell.r, r); // cell.r == r
555
556           cell.mark = false;
557         }
558
559       // Searching starting by row. Must be equivalent to starting from column.
560       VERIFY(find_loop(rows, cols, goalR, goalC, goalR, goalC, idxsOut, true, 1));
561     }
562   } // End of anonymous namespace.
563
564   template< class M >
565   typename M::value_type transportationCost( matrix_expression<M> const &C,
          matrix_expression<M> const &X )
566   {
567     typedef typename M::value_type scalar_type;
568
569     ASSERT_EQ(C().size1(), X().size1());
570     ASSERT_EQ(C().size2(), X().size2());
571
572     size_t m = C().size1(), n = C().size2();
573
574     scalar_type result = scalar_type();
575     // TODO: Use ublas functions.
576     for (size_t r = 0; r < m; ++r)
577       for (size_t c = 0; c < n; ++c)
578         result += C()(r, c) * X()(r, c);
579
580     return result;
581   }
582
583   template< class V1, class V2, class M1, class M2 >
584   void solve( vector_expression<V1> const &a,
585               vector_expression<V2> const &b,
586               matrix_expression<M1> const &C,
587               matrix_expression<M2>       &X )
588   {
589     typedef typename V1::value_type scalar_type; // TODO
590     typedef vector<scalar_type>       vector_type;
591     typedef zero_vector<scalar_type> zero_vector_type;
592     typedef matrix<scalar_type>       matrix_type;
593     typedef zero_matrix<scalar_type> zero_matrix_type;
594
595     ASSERT(assert_tp_valid(a, b, C));
596     ASSERT(is_tp_closed(a, b, C));
597
598     size_t const m = a().size(), n = b().size();
599
600     // Building start plan.
601     matrix_type x;
602     all_cells_map_type planCells;
603
604     build_start_plan(a, b, C, x, planCells);
```

```
605
606       // Building structure for fast plan cells retrieving.
607       cells_maps_vector_type rows(m), cols(n);
608       for (all_cells_map_type::const_iterator it = planCells.begin(); it != planCells.end()
              ; ++it)
609       {
610         size_t const r = it->first.first, c = it->first.second;
611         cell_ptr_type cellPtr = it->second;
612
613         ASSERT_EQ(r, cellPtr->r);
614         ASSERT_EQ(c, cellPtr->c);
615
616         VERIFY(rows[r].insert(std::make_pair(c, cellPtr)).second);
617         VERIFY(cols[c].insert(std::make_pair(r, cellPtr)).second);
618       }
619
620       // Iterating through all plans.
621       vector_type u, v;
622       matrix_type P;
623       size_t const nMaxIterations(1000); // debug
624       size_t nIterations(0);
625       scalar_type prevPlanTransportationCost = transportationCost(C, x);
626       while (true)
627       {
628         // Recalculating potentials coefficients.
629         calculate_potentials_coefs(rows, cols, C, u, v);
630
631         // Recalculating potentials.
632         calculate_potentials(u, v, C, P);
633
634         // Searching cell with maximum potential.
635         size_t maxPRow(0), maxPColumn(0);
636         for (size_t r = 0; r < m; ++r)
637           for (size_t c = 0; c < n; ++c)
638             if (P(maxPRow, maxPColumn) < P(r, c))
639             {
640               maxPRow    = r;
641               maxPColumn = c;
642             }
643
644         if (!eq(x(maxPRow, maxPColumn), scalar_type()))
645         {
646           // Found optimal plan. Interrupting.
647           break;
648         }
649         ASSERT_EQ(x(maxPRow, maxPColumn), scalar_type());
650
651         // Searching loop.
652         std::vector<std::pair<size_t, size_t> > loopCellsIdxs;
653         {
654           std::vector<std::pair<size_t, size_t> > reversedLoopCellsIdxs;
655           find_loop(rows, cols, maxPRow, maxPColumn, std::back_inserter(
                  reversedLoopCellsIdxs));
656           ASSERT_GE(reversedLoopCellsIdxs.size(), 4);
657           ASSERT_LE(reversedLoopCellsIdxs.size(), m + n - 1);
658           ASSERT_EQ(reversedLoopCellsIdxs.size() % 2, 0);
659           ASSERT(reversedLoopCellsIdxs[reversedLoopCellsIdxs.size() - 1] == std::make_pair(
                  maxPRow, maxPColumn));
660
661           loopCellsIdxs.assign(reversedLoopCellsIdxs.rbegin(), reversedLoopCellsIdxs.rend()
                  );
662         }
663
664         // Locating cell from which shipment will be canceled on even subloop.
665         std::pair<size_t, scalar_type> minShipment = std::make_pair(1, x(loopCellsIdxs[1].
              first, loopCellsIdxs[1].second));
666         for (size_t i = 1 + 2; i < loopCellsIdxs.size(); i += 2)
667         {
668           scalar_type const curShipment = x(loopCellsIdxs[i].first, loopCellsIdxs[i].second
                  );
669           if (curShipment < minShipment.second)
```

```
670           {
671             minShipment.first = i;
672             minShipment.second = curShipment;
673           }
674         }
675
676         // Minimum shipment on loop.
677         size_t const minShipmentR = loopCellsIdxs[minShipment.first].first;
678         size_t const minShipmentC = loopCellsIdxs[minShipment.first].second;
679
680         scalar_type const shipmentValue = minShipment.second;
681
682         // Removing on even subloop.
683         for (size_t i = 1; i < loopCellsIdxs.size(); i += 2)
684         {
685           size_t const r = loopCellsIdxs[i].first;
686           size_t const c = loopCellsIdxs[i].second;
687           x(r, c) -= shipmentValue;
688         }
689
690         // Adding on odd subloop.
691         for (size_t i = 0; i < loopCellsIdxs.size(); i += 2)
692         {
693           size_t const r = loopCellsIdxs[i].first;
694           size_t const c = loopCellsIdxs[i].second;
695           x(r, c) += shipmentValue;
696         }
697
698         // Asserting that new plan is a plan.
699         ASSERT(is_plan(a, b, x));
700
701         // Asserting that it cost is lower than cost of previus plan.
702         scalar_type const newPlanTransportationCost = transportationCost(C, x);
703         ASSERT_LE(newPlanTransportationCost, prevPlanTransportationCost);
704         prevPlanTransportationCost = newPlanTransportationCost;
705
706         // Updating storage: removing minimum shipment variable and adding maximum
              potential variable.
707
708         // Removing from storage.
709         VERIFY_EQ(rows[minShipmentR].erase(minShipmentC), 1);
710         VERIFY_EQ(cols[minShipmentC].erase(minShipmentR), 1);
711         VERIFY_EQ(planCells.erase(std::make_pair(minShipmentR, minShipmentC)), 1);
712
713         // Adding to storage.
714         cell_ptr_type newCell(new cell_type(maxPRow, maxPColumn));
715         VERIFY(planCells.insert(std::make_pair(std::make_pair(maxPRow, maxPColumn), newCell
              )).second);
716         VERIFY(cols[maxPColumn].insert(std::make_pair(maxPRow, newCell)).second);
717         VERIFY(rows[maxPRow].insert(std::make_pair(maxPColumn, newCell)).second);
718
719         // debug
720         ++nIterations;
721         if (nIterations > nMaxIterations)
722           std::cerr << "lp_potentials::solve():␣too␣much␣iterations!" << std::endl;
723         // end of debug
724       }
725
726       // Copying result matrix.
727       X() = x;
728     }
729 } // End of namespace 'lp_potentials'.
730 } // End of namespace 'numeric'.
731
732 #endif // NUMERIC_POTENTIALS_ALG_HPP
```