RogueWave SOFTWARE

Accelerating Great Code

A FAST, SCALABLE SOLUTION FOR SOLVING THE TRANSPORTATION PROBLEM

The IMSL C Numerical Library version 2016.0 includes a function to solve the transportation problem. This code existed previously only in the IMSL Fortran library as `TRAN()`. Given a recent customer request, the development team has ported the algorithm to C as `imsl_f_transport()`, including performance tuning using OpenMP directives. This paper describes the problem, provides an overview of the solution technique, and covers some similar problems that can be solved with the same function. Several examples are provided to illustrate the different problem types.

THE TRANSPORTATION PROBLEM DESCRIPTION

The transportation problem can be described using examples from many fields. Apparently, the original use case was a challenge of efficiently moving troops from bases to battleground locations; however, the more common use is of moving goods from multiple factories to multiple warehouse locations, or from warehouses to storefronts. Before diving into the mathematics, consider the following example:

> XYZ Inc. has two factories in different locations around the country where they produce widgets. Their sales partner has three central warehouses where they ship these widgets to their various customers. The factories can produce a given number of widgets per week each and the expected demand for each warehouse is also known. There is a shipping cost from each factory to each warehouse. Which factory should produce and ship how many widgets to which warehouses to meet the demand at each location with minimal cost?

This problem statement is typical of transportation problems. Before diving into a specific example with numbers, ponder the translation into mathematical expressions. The source and destinations are irrelevant — they could be logging sites and saw mills, factories and warehouses, warehouses and stores, troops and battlefields, and so on. In each case there is some demand *D* at *N* locations and some supply *S* at *M* locations with a cost matrix for each of the *M* x *N* paths. Again, the cost, *c*, could be a dollar cost, or time, or some calculation involving multiple factors, so its units are irrelevant in the problem statement. The solution is the number of units, *x*, to be produced at each supply center and sent to each demand location. The total cost is then the unit cost for each path, c_{ij}, times the number associated with that path, x_{ij}. Framing this as an optimization problem, the goal is to minimize:

$$\sum_{i=1}^{M}\sum_{j=1}^{N} c_{ij}x_{ij}$$

There are some constraints too, of course. The number of units shipped must be less than or equal to the total supply. Similarly, the number shipped has to match the demand at each location. Finally, the number of units to ship must be greater than or equal to zero (no negative values). For a balanced transportation problem, the total demand is equal to the total supply, leading to the following constraints:

$$\sum_{j=1}^{N} x_{ij} = S_i, i = 1, \dots, M$$

$$\sum_{i=1}^{M} x_{ij} = D_j, j = 1, \dots, N$$

$$x_{ij} \geq 0, i = 1 \dots, M, j = 1 \dots N$$

There may be cases of excess supply or excess demand leading to an unbalanced problem; this case is addressed below and can be solved similarly using dummy variables and possibly penalties for unmet demand or storage costs for excess supply.

The combination of the equation to minimize and the three constraints define a well-formed linear programming (LP) optimization problem with linear constraints, specifically the balanced transportation problem.

THE SOLUTION

Unlike many LP problems, the transportation problem is feasible to solve by hand using a series of tables and well-documented strategies such as the Northwest-Corner Method to find an initial basic feasible solution and then using techniques like the Least-Cost Method or the Stepping Stone Method. Working through a problem like this by hand can be a rewarding experience, which simply requires repeatedly drawing tables and manual refinement starting with an initial basic feasible solution. However, for larger problems or to realize any level of scalability, a computer-based method is preferred. Spreadsheet solutions are possible as well, but often require significant rework whenever the number of supply or demand centers changes.

The IMSL Library algorithm skips the paperwork and allows users to solve problems of nearly any size with a simple programming interface. The IMSL C function is `imsl_f_transport()` with a short list of required arguments and a few optional arguments. Using the terminology introduced above, the calling sequence using only the required arguments is:

```
z = imsl_f_transport(M, N, S, D, &c[0][0], 0);
```

where each variable is defined as:

z = a pointer to the solution matrix of size M x N containing the optimal routing
M = the number of sources (supply locations)
N = the number of sinks (demand locations)
S = an array of size M containing the supply capacities
D = an array of size N containing the demand requirements
c = an array of size M x N containing the per-unit cost matrix

Optional arguments provide additional capabilities such as limiting the number of iterations (by default there is no limit) and output of the dual solution and total cost of the optimal routing.

The algorithm used is a revised simplex method, solving a very sparse LP problem with M + N constraints and M x N variables. The solution requires integer values since a partial unit cannot be shipped, so strictly speaking this integer constraint means this is not a traditional LP problem. However, a special integer programming algorithm is not required because the transportation matrix is unimodular. If the supply capacities and demand requirements are all integer, then the modularity of the M x N transportation matrix guarantees that every basic feasible solution is integer. Since every optimal solution of the transportation problem is a basic feasible solution, it is also integer. Therefore, every algorithm that works with basic feasible solutions will find an integer solution (for example the case for different variants of the simplex method). Internally to the simplex algorithm, division is required during the pivot calculation, but in this problem type, the element of the pivot will always be one, resulting in only integer solutions. Of course, if some of the supply or demand values are not integer values, then non-integer solutions are possible.

AN EXAMPLE

Consider the word problem stated above again, but this time with some actual values.

> XYZ Inc. has two factories with weekly production rates of 40 and 20 widgets. These widgets must be shipped to three warehouses that have a demand of 25, 10, and 25 units. The cost to ship between each location is known (see the grid below). Which factory should produce and ship how many widgets to which warehouses to meet the demand at each location with minimal cost?

The standard table format of this problem is:

($ cost)	**Warehouse 1**	**Warehouse 2**	**Warehouse 3**	Supply
Factory A	550	300	400	40
Factory B	350	300	100	20
Demand	25	10	25	

This table format is used throughout this paper for both the problem statement and solutions. When the upper leftmost cell is "(solution)" the contents are the solution matrix, otherwise it's a cost matrix of specified units.

The central values in the table represent the point-to-point per unit shipping cost, such that it costs $550 to ship from Factory A to Warehouse 1. Note that this is a balanced problem where the total supply of 60 units equals the total demand. The C code to solve this problem definition using the IMSL algorithm is as follows, including the optional output parameter to display the total cost:

```
#include <stdio.h>
#include <imsl.h>

#define      NF    2
#define      NW    3

int main() {
    int i, j;
    float cmin, *x;
    float fcap[NF] = { 40, 20 };
    float wdem[NW] = { 25, 10, 25 };
    float cost[NF][NW] = {
        { 550, 300, 400 },
        { 350, 300, 100 }
    };

    x = imsl_f_transport(NF, NW, fcap, wdem, &cost[0][0],
        IMSL_TOTAL_COST, &cmin, 0);

    printf("Minimum cost is $%.2f\n", cmin);
    for (i = 0; i < NF; i++) {
        for (j = 0; j < NW; j++) {
            printf("Ship %3.0f units from factory %c to warehouse %d\n",
                x[i * NW + j], i + 65, j + 1);
        }
    }

    imsl_free(x);
    return 1;
}
```

Executing the program produces the following output:

```
Minimum cost is $20750.00

Ship 25 units from factory A to warehouse 1
Ship 10 units from factory A to warehouse 2
Ship  5 units from factory A to warehouse 3
Ship  0 units from factory B to warehouse 1
Ship  0 units from factory B to warehouse 2
Ship 20 units from factory B to warehouse 3
```

Using the previous table format, replacing unit costs with the number of units gives:

(solution)	**Warehouse 1**	**Warehouse 2**	**Warehouse 3**	Supply
Factory A	25	10	5	40
Factory B	0	0	20	20
Demand	25	10	25	

This table view allows easy confirmation that the supply for each factory and the demand at each warehouse is fully satisfied.

UNBALANCED PROBLEMS

Not every transportation is necessarily balanced. When supply exceeds demand, an alternate technique is not essential, especially if there are no storage costs associated with not sending units to the destination. However, when there is excess demand, the problem statement is infeasible as there is no solution that can satisfy the demand at each destination. The IMSL algorithm will issue a warning of IMSL_INSUFFICIENT_CAPACITY if this condition is encountered, although it will still return the best solution possible. When attempting to solve an unbalanced problem, best practices dictate the use of dummy variables to take up the slack.

Consider an example of allocating rental cars. For simplicity, generic locations will be used and the cars are assumed to be fungible (any car can be substituted for any other car). There is also a storage cost to keep an unwanted car at an origin location which must be accounted for. The base problem given is: Sites A, B, and C have 8, 12, and 10 cars on site, while Destinations X, Y, and Z require 9, 7, and 11 cars respectively. The storage cost for each site is 100, 100, and 80. The cost to move the cars between sites is shown in the table:

($ cost)	**Dest X**	**Dest Y**	**Dest Z**	Supply
Site A	460	350	640	8
Site B	510	420	690	12
Site C	650	680	490	10
Demand	9	7	11	

From this problem statement, it is clear that demand (27 cars) is less than the supply (30 cars), so three cars will need to be kept at one or more source locations. To account for this, one would add a dummy destination site with the appropriate storage cost:

($ cost)	Dest X	Dest Y	Dest Z	Dummy	Supply
Site A	460	350	640	100	8
Site B	510	420	690	100	12
Site C	650	680	490	80	10
Demand	9	7	11	3	

In this form, the problem is balanced. Except for now including the storage cost and location of the extra supply, the solution returned from the IMSL transport function is the same in either case, thus there is no strict requirement in the algorithm for the problem to be balanced. The source code for this problem is included in Appendix A, with the solution matrix as follows, and a total cost of $12,880:

(solution)	Dest X	Dest Y	Dest Z	Stored	Supply
Site A	0	7	1	0	8
Site B	9	0	0	3	12
Site C	0	0	10	0	10
Demand	9	7	11	3	

Alternatively, if a problem indicates an excess demand, a similar dummy column can be utilized, especially if there is a known cost for missing the demand (like lost revenue from dissatisfied customers). Sometimes there is no cost associated with missing demand or excess supply, and in those cases the value in the cost matrix for the dummy row or column would simply be zero.

There may also be special cases where one or more destinations cannot be reached from one or more source locations. In this case, the user should use an arbitrarily large number in the cost matrix to encourage the algorithm to allocate zero units for this unreachable condition. A value an order of magnitude or two larger than any other cost in the matrix usually suffices.

THE ASSIGNMENT PROBLEM

The assignment problem is a special case of the transportation problem. In this problem, each of the supply items maps to one and only one of the demand items. This type of problem appears in examples such as assigning tasks or even choosing a relay team. As the 2016 Summer Olympics have recently completed, the next example builds the best swimming medley relay team given the four events and the times of five swimmers for each event. The solution will have one and only one swimmer assigned to each leg of the event. The main difference from the general transportation is that each supply item (a swimmer) and each destination item (a leg of the race) is only one unit, as seen in the following table:

(time*10 sec)	Back	Breast	Fly	Free	Supply
Swimmer A	479	484	477	474	1
Swimmer B	481	477	481	467	1
Swimmer C	486	474	479	476	1
Swimmer D	483	482	486	480	1
Swimmer E	485	483	484	479	1
Demand	1	1	1	1	

Note that swim times are measured to the tenths of a second (for example 48.1 seconds); to keep everything integer, the times have been multiplied by ten. Due to the linearity of the problem, scaling the cost matrix has no effect on the solution. With unit supply and demand across the board, the resulting solution will be mostly zeros with a value of one located for the best choice of swimmer for each leg of the medley relay.

No special formulation is required when using the IMSL transport function to solve the assignment problem. The code is available in Appendix B for this problem. The solution matrix is below, giving an optimal best time of 190.1 seconds.

(solution)	**Back**	**Breast**	**Fly**	**Free**	Supply
Swimmer A	0	0	0	1	1
Swimmer B	0	0	1	0	1
Swimmer C	0	1	0	0	1
Swimmer D	1	0	0	0	1
Swimmer E	0	0	0	0	1
Demand	1	1	1	1	

THE TRANSSHIPMENT PROBLEM

Another special case of the transportation problem is the transshipment problem. This scenario arises when intermediary destinations are more cost effective (such as shipping from Los Angeles to New York via Denver may be less expensive than shipping direct) or when additional modes of transportation are available at different costs (shipping via truck versus rail).

The table layout continues to be useful to illustrate this problem. In short, additional points (the transshipment points) need to be appended to both the columns and rows, with each supply and demand value equal to the total supply and demand (assuming, of course, the problem is balanced or has been balanced with dummy variables). In the cost matrix, the point-to-point costs are all still required, except that there will be zero values for the cost where the same label exists in both the supply and demand. Arbitrarily large values are used for any paths that aren't logistically valid, for example shipping between transshipment points.

Consider an example from the book Introduction to Management Science[1] where wheat is harvested in Nebraska and Colorado, then must be shipped to grain elevators in Kansas City, Omaha, and Des Moines. These transshipment points then send the product along to Chicago, St. Louis, and Cincinnati. The farms have outputs of 300 units each, while the demand is 200, 100, and 300 respectively at each destination. In this problem statement, the farms cannot ship to the endpoints directly, and the transshipment points cannot ship to each other. Each of the point-to-point shipping costs appear in the table below. Wherever a path is not valid, a cost of 1000 is applied. In this case, the farms cannot ship directly to the destinations, and the transshipment points are not allowed to transfer between each other. For the transshipment points, which appear as both sources and destinations, there is zero cost to ship to itself. Using the table format again to formulate the problem and provide the point-to-point shipping costs gives:

[1] Taylor, Bernard W. III (2009) *Introduction to Management Science (10th Edition)*. New York, NY: Pearson.

($ cost)	Chicago	St. Louis	Cincinnati	Kansas City	Omaha	Des Moines	Supply
Nebraska	1000	1000	1000	16	10	12	300
Colorado	1000	1000	1000	15	14	17	300
Kansas City	6	8	10	0	1000	1000	600
Omaha	7	11	11	1000	0	1000	600
Des Moines	4	5	12	1000	1000	0	600
Demand	200	100	300	600	600	600	

The problem is balanced, noting that the transshipment points each have a supply and demand equal to the sum of the actual values, 600. Source code for this problem is located in Appendix C. The solution matrix is shown below for a total cost of $12,400:

(solution)	Chicago	St. Louis	Cincinnati	Kansas City	Omaha	Des Moines	Supply
Nebraska	0	0	0	0	0	300	300
Colorado	0	0	0	300	0	0	300
Kansas City	0	0	300	300	0	0	600
Omaha	0	0	0	0	600	0	600
Des Moines	200	100	0	0	0	300	600
Demand	200	100	300	600	600	600	

The solution indicates that the 300 units from Nebraska are shipped to Des Moines, while all 300 units from Colorado are shipped to Kansas City. Once in Kansas City, all 300 units are sent off to Cincinnati, but at Des Moines, the shipment is split with 200 heading to Chicago and 100 to St. Louis. Notice the 600 units in Omaha indicate that this point is not part of the solution as the site is meeting its (artificial) demand with its own (artificial) supply. Once the complexity of the problem is pared down and understood in table format, translating the data for solving with IMSL is relatively straightforward as the cost matrix ordering and supply/demand values can be plucked directly from the table.

SUMMARY

While the family of transportation problems can be solved by hand, at least for relatively small problems, the IMSL Library includes an algorithm that is fast and scalable. The IMSL Fortran Library `TRAN()` function has been ported to the IMSL C Library family as `imsl_f_transportation()`, which is the focus of the discussion and examples provided here. To discover more information on the IMSL algorithms discussed here, visit IMSL Numerical Libraries.

APPENDIX

A. Source code for unbalanced problem example

```
#include <stdio.h>
#include <imsl.h>

#define NS 3
#define ND 4

int main() {
      float cmin, *x;
      float sup[NS] = { 8, 12, 10 };
      float dem[ND] = { 9, 7, 11, 3 };
      float cost[NS][ND] = {
            { 460, 350, 640, 100 },
            { 510, 420, 690, 100 },
            { 650, 680, 490, 80 }
      };

      x = imsl_f_transport(NS, ND, sup, dem, &cost[0][0],
            IMSL_TOTAL_COST, &cmin, 0);

      printf("Minimum cost is $%.2f\n", cmin);

      imsl_f_write_matrix("Solution Matrix", NS, ND, x,
            IMSL_NO_ROW_LABELS, IMSL_NO_COL_LABELS, 0);

      imsl_free(x);
      return 1;
}
```

B. Source code for assignment problem example

```
#include <stdio.h>
#include <imsl.h>

#define NS 5
#define ND 4

int main() {
      float cmin, *x;
      float sup[NS] = { 1, 1, 1, 1, 1 };
      float dem[ND] = { 1, 1, 1, 1 };
      float cost[NS][ND] = {
            { 479, 484, 477, 474 },
            { 481, 477, 481, 467 },
            { 486, 474, 479, 476 },
            { 483, 482, 486, 480 },
            { 485, 483, 484, 479 }
```

```
      };
      x = imsl_f_transport(NS, ND, sup, dem, &cost[0][0],
            IMSL_TOTAL_COST, &cmin, 0);

      printf("Minimum time is %.2f sec", cmin/10);

      imsl_f_write_matrix("Solution Matrix", NS, ND, x,
            IMSL_NO_ROW_LABELS, IMSL_NO_COL_LABELS, 0);

      imsl_free(x);
      return 1;
}
```

C. Source code for transshipment problem example

```
#include <stdio.h>
#include <imsl.h>

#define NS 5
#define ND 6

int main() {
      float cmin, *x;
      float sup[NS] = { 300, 300, 600, 600, 600 };
      float dem[ND] = { 200, 100, 300, 600, 600, 600 };
      float cost[NS][ND] = {
            { 1000, 1000, 1000,   16,   10,   12 },
            { 1000, 1000, 1000,   15,   14,   17 },
            {    6,    8,   10,    0, 1000, 1000 },
            {    7,   11,   11, 1000,    0, 1000 },
            {    4,    5,   12, 1000, 1000,    0 }
      };

      x = imsl_f_transport(NS, ND, sup, dem, &cost[0][0],
            IMSL_TOTAL_COST, &cmin, 0);

      printf("Minimum cost is $%.2f", cmin);

      imsl_f_write_matrix("Solution Matrix", NS, ND, x,
            IMSL_NO_ROW_LABELS, IMSL_NO_COL_LABELS, 0);

      imsl_free(x);
      return 1;
}
```

Rogue Wave provides software development tools for mission-critical applications. Our trusted solutions address the growing complexity of building great software and accelerates the value gained from code across the enterprise. The Rogue Wave portfolio of complementary, cross-platform tools helps developers quickly build applications for strategic software initiatives. With Rogue Wave, customers improve software quality and ensure code integrity, while shortening development cycle times. roguewave.com