USER'S GUIDE FOR TOMLAB /LDO[1]

Kenneth Holmström[2], Anders O. Göran[3] and Marcus M. Edvall[4]

February 14, 2005

[1]More information available at the TOMLAB home page: `http://tomlab.biz` and at the Applied Optimization and Modeling **TOM** home page `http://www.ima.mdh.se/tom`. E-mail: **tomlab@tomlab.biz**.

[2]Professor in Optimization, Mälardalen University, Department of Mathematics and Physics, P.O. Box 883, SE-721 23 Västerås, Sweden, kenneth.holmstrom@mdh.se.

[3]Tomlab Optimization AB, Västerås Technology Park, Trefasgatan 4, SE-721 30 Västerås, Sweden, anders@tomlab.BIZ.

[4]Tomlab Optimization Inc., 855 Beech St #121, San Diego, CA, USA, medvall@tomlab.BIZ.

Contents

Contents **2**

1 TOMLAB LDO (Linear and Discrete Optimization) **4**

1.1 Optimization Algorithms and Solvers in TOMLAB LDO . . . 4
1.1.1 Linear Programming . . . 4
1.1.2 Transportation Programming . . . 5
1.1.3 Network Programming . . . 6
1.1.4 Mixed-Integer Programming . . . 6
1.1.5 Dynamic Programming . . . 6
1.1.6 Quadratic Programming . . . 7
1.1.7 Lagrangian Relaxation . . . 7
1.1.8 Utility Routines . . . 8
1.2 How to Solve Optimization Problems Using TOMLAB LDO . . . 9
1.2.1 How to Solve Linear Programming Problems . . . 9
1.2.2 How to Solve Transportation Programming Problems . . . 10
1.2.3 How to Solve Network Programming Problems . . . 11
1.2.4 How to Solve Integer Programming Problems . . . 13
1.2.5 How to Solve Dynamic Programming Problems . . . 14
1.2.6 How to Solve Lagrangian Relaxation Problems . . . 15
1.3 Printing Utilities and Print Levels . . . 16
1.4 Optimization Routines in TOMLAB LDO . . . 17
1.4.1 akarmark . . . 17
1.4.2 balas . . . 18
1.4.3 dijkstra . . . 19
1.4.4 dpinvent . . . 20
1.4.5 dpknap . . . 21
1.4.6 karmark . . . 22
1.4.7 ksrelax . . . 23
1.4.8 labelcor . . . 24
1.4.9 lpdual . . . 25
1.4.10 lpkarma . . . 27
1.4.11 lpsimp1 . . . 28
1.4.12 lpsimp2 . . . 29
1.4.13 maxflow . . . 30

1.4.14 modlabel . . . 31
1.4.15 NWsimplx . . . 32
1.4.16 qplm . . . 33
1.4.17 qpe . . . 34
1.4.18 salesman . . . 35
1.4.19 TPsimplx . . . 36
1.4.20 urelax . . . 38
1.5 Optimization Subfunction Utilities in TOMLAB LDO . . . 39
1.5.1 a2frstar . . . 39
1.5.2 gsearch . . . 40
1.5.3 gsearchq . . . 41
1.5.4 mintree . . . 42
1.5.5 TPmc . . . 43
1.5.6 TPnw . . . 44
1.5.7 TPvogel . . . 45
1.5.8 z2frstar . . . 46

References **47**

1 TOMLAB LDO (Linear and Discrete Optimization)

TOMLAB LDO is a collection of routines in TOMLAB for solving linear and discrete optimization (LDO) problems in operations research and mathematical programming. Included are many routines for special problems in linear programming, network programming, integer programming and dynamic programming.

Note that included in standard TOMLAB are the standard solver *lpSolve* for linear programming, the solver *DualSolve*, used to solve linear programming problems when a dual feasible point is known, and two routines for mixed-integer programming, *mipSolve* and *cutplane*. The aim of the corresponding simpler routines in the LDO collection are mainly teaching. Use the standard TOMLAB routines for production runs.

1.1 Optimization Algorithms and Solvers in TOMLAB LDO

This section describes the LDO routines by giving tables describing most Matlab functions with some comments. All function files are collected in the directory *ldo*.

There is a simple menu program, *ShowSimplex*, for linear programming. The routine is a utility to interactively solve LP problems in canonical standard form. When the problem is defined, *ShowSimplex* calls the TOMLAB internal LDO solvers *lpsimp1* and *lpsimp2*.

Like the MathWorks, Inc. Optimization Toolbox 1.x, TOMLAB LDO is using a vector with optimization parameters. In Optimization Toolbox, the routine setting up the default values in a vector OPTIONS with 18 parameters is called *foptions*. Our solvers need more parameters, currently 29, and therefore the routine *goptions* is used instead of *foptions*.

In TOMLAB the routine *lpDef* is used to define the *optPar* vector and the routine *optParamDef* the *optParam* structure.

1.1.1 Linear Programming

There are several algorithms implemented for **linear programming**, listed in Table 1. The solver *lpSolve2* is using the same input and output format as the TOMLAB solvers described in the TOMLAB User's guide. It is using the optimization parameter structure *optParam* instead of the optimization parameter vector *optPar*.

lpsimp1 and *lpsimp2* are simpler versions of the two basic parts in *lpSolve2* that solves Phase I and Phase II LP problem, respectively. *lpdual* is an early version of the TOMLAB solver *DualSolve*.

lpSolve calls both the routines *Phase1Simplex* and *Phase2Simplex* to solve a general **linear program** (**lp**) defined as

$$\begin{array}{cl} \min\limits_{x} & f(x) = c^T x \\ \\ s/t & \begin{array}{lcccl} x_L & \leq & x & \leq & x_U, \\ b_L & \leq & Ax & \leq & b_U \end{array} \end{array} \tag{1}$$

where $c, x, x_L, x_U \in \mathbb{R}^n$, $A \in \mathbb{R}^{m_1 \times n}$, and $b_L, b_U \in \mathbb{R}^{m_1}$.

The implementation of *lpsimp2* is based on the standard revised simplex algorithm as formulated in Goldfarb and Todd [5, page 91] for solving a Phase II LP problem. *lpsimp1* implements a Phase I simplex strategy which formulates a LP problem with artificial variables. This routine is using *lpsimp2* to solve the Phase I problem. The dual simplex method [5, pages 105-106], usable when a dual feasible solution is available instead of a primal feasible, is also implemented (*lpdual*).

Table 1: Solvers for linear programming.

Function	Description	Section	Page
lpSolve2	General solver for linear programming problems. Has two internal routines. Phase1Simplex finds basic feasible solution (bfs) using artificial variables. It calls the other internal routine, Phase2Simplex, which implements the revised simplex algorithm with three selection rules.		
akarmark	Affine scaling variant of Karmarkar's algorithm.	1.4.1	17
karmark	Karmakar's algorithm. Kanonical form.	1.4.6	22
lpdual	The dual simplex algorithm.	1.4.9	25
lpkarma	Solves LP on equality form, by converting and calling *karmark*.	1.4.10	27
lpsimp1	The Phase I simplex algorithm. Finds a basic feasible solution (bfs) using artificial variables. Calls *lpsimp2*.	1.4.11	28
lpsimp2	The Phase II revised simplex algorithm with three selection rules.	1.4.12	29

Two polynomial algorithms for linear programming are implemented. Karmakar's projective algorithm (*karmark*) is developed from the description in Bazaraa et. al. [3, page 386]. There is a choice of update, either according to Bazaraa or the rule by Goldfarb and Todd [5, chap. 9]. The affine scaling variant of Karmakar's method (*akarmark*) is an implementation of the algorithm in Bazaraa [5, pages 411-413]. As the purification algorithm a modification of the algorithm on page 385 in Bazaraa is used.

The internal linear programming solvers *lpsimp2* and *lpdual* both have three rules for variable selection implemented. Bland's cycling prevention rule is the choice if fear of cycling exists. There are two variants of minimum reduced cost variable selection, the original Dantzig's rule and one which sorts the variables in increasing order in each step (the default choice).

1.1.2 Transportation Programming

Transportation problems are solved using an implementation of the transportation simplex method as described in Luenberger [8, chap 5.4] (*TPsimplx*). Three simple algorithms to find a starting basic feasible solution for the transportation problem are included; the northwest corner method (*TPnw*), the minimum cost method (*TPmc*) and Vogel's approximation method (*TPvogel*). The implementation of these algorithms follows the algorithm descriptions in Winston [9, chap. 7.2]. The functions are described in Table 2.

Table 2: Routines for transportation programming.

Function	Description	Section	Page
TPsimplx	Implementation of the transportation simplex algorithm.	1.4.19	36
TPmc	Find initial bfs to TP using the minimum cost method.	1.5.5	43
TPnw	Find initial bfs to TP using the northwest corner method.	1.5.6	44
TPvogel	Find initial bfs to TP using Vogel's approximation method.	1.5.7	45

1.1.3 Network Programming

The implementation of the **network programming** algorithms are based on the forward and reverse star representation technique described in Ahuja et al. [2, pages 35-36]. The following algorithms are currently implemented:

- Solve the shortest path problem using Dijkstra's algorithm (*dijkstra*). A direct implementation of the Algorithm DIJKSTRA in [1, pages 250-251].
- Solve the shortest path problem using a label correcting method (*labelcor*). The implementation is based on Algorithm LABEL CORRECTING in [1, page 260].
- Solve the maximum flow problem using the Ford-Fulkerson augmenting path method (*maxflow*). The implementation is based on the algorithm description in Luenberger [8, pages 144-145].
- Solve the shortest path problem using a modified label correcting method (*modlabel*). The implementation is based on Algorithm MODIFIED LABEL CORRECTING in [1, page 262], including the heuristic rule discussed to improve running time in practice.
- Solve the minimum cost network flow problem (MCNFP) using a network simplex algorithm (*NWsimplx*). The implementation is based on Algorithm network simplex in Ahuja et. al. [2, page 415].
- Search for all reachable nodes in a network using a stack approach (*gsearch*). The implementation is a variation of the Algorithm SEARCH in [1, pages 231-233].
- Solve the symmetric traveling salesman problem using Lagrangian relaxation and the subgradient method with the Polyak rule II (*salesman*), an algorithm by Held and Karp [6].
- Search for all reachable nodes in a network using a queue approach (*gsearchq*). The implementation is a variation of the Algorithm SEARCH in [1, pages 231-232].
- Find the minimal spanning tree of an undirected graph (*mintree*) with Kruskal's algorithm described in Ahuja et. al. [2, page 520-521].

The network programming routines are listed in Table 3.

1.1.4 Mixed-Integer Programming

To solve mixed linear inequality integer programs two algorithms are implemented as part of TOMLAB, *mipSolve* and *cutplane*. Described in the Network Programming section 1.1.3 is the *salesman* routine, which is an special type of integer programming problem. The directory *tsplib* contains test problems for the travelling salesman problem. The routine *runtsp* runs any of the 25 predefined problems. *tsplib* reads the actual problem definition and generates the problem.

Balas method for binary integer programs restricted to integer coefficients is implemented in the routine *balas* [7].

1.1.5 Dynamic Programming

Two simple examples of dynamic programming are included. Both examples are from Winston [9, chap. 20]. Forward recursion is used to solve an inventory problem (*dpinvent*) and a knapsack problem (*dpknap*), see Table 4.

Table 3: Routines for network programs.

Function	Description	Section	Page
dijkstra	Shortest path using Dijkstra's algorithm.	1.4.3	19
labelcor	Shortest path using a label correcting algorithm.	1.4.8	24
maxflow	Solving maximum flow problems using the Ford-Fulkerson augmenting path method.	1.4.13	30
modlabel	Shortest path using a modified label correcting algorithm.	1.4.14	31
NWsimplx	Solving minimum cost network flow problems (MCNFP) with a network simplex algorithm.	1.4.15	32
salesman	Symmetric traveling salesman problem (TSP) solver using Lagrangian relaxation and the subgradient method with the Polyak rule II.	1.4.18	35
gsearch	Searching all reachable nodes in a network. Stack approach.	1.5.2	40
gsearchq	Searching all reachable nodes in a network. Queue approach.	1.5.3	41
mintree	Finds the minimum spanning tree of an undirected graph.	1.5.4	42

Table 4: Routines for dynamic programming.

Function	Description	Section	Page
dpinvent	Forward recursion DP algorithm for the inventory problem.	1.4.4	20
dpknap	Forward recursion DP algorithm for the knapsack problem.	1.4.5	21

1.1.6 Quadratic Programming

Two simple routines for quadratic programming are included,

Table 5: Routines for quadratic programming.

Function	Description	Section	Page
qplm	Solves a **qp** problem, restricted to equality constraints, using Lagrange's method.	1.4.16	33
qpe	Solves a **qp** problem, restricted to equality constraints, using a null space method.	1.4.17	34

1.1.7 Lagrangian Relaxation

The usage of Lagrangian relaxation techniques is exemplified by the routine *ksrelax*, which solves integer linear programs with linear inequality constraints and upper and lower bounds on the variables. The problem is solved by relaxing all but one constraint and hence solving simple knapsack problems as subproblems in each iteration. The algorithm is based on the presentation in Fischer [4], using subgradient iterations and a simple line search rule. Lagrangian relaxation is used by the symmetric travelling salesman solver *salesman*. Also a routine to draw a plot of the relaxed function is included. The Lagrangian relaxation routines are listed in Table 6.

Table 6: Routines for Lagrangian relaxation.

Function	Description	Section	Page
ksrelax	Lagrangian relaxation with knapsack subproblems.	1.4.7	23
urelax	Lagrangian relaxation with knapsack subproblems, plot result.	1.4.20	38

1.1.8 Utility Routines

Table 7 lists the utility routines used in TOMLAB LDO. Some of them are also used by the other routines in TOMLAB.

Table 7: Utility routines.

Function	Description
a2frstar	Convert node-arc A matrix to Forward-Reverse Star Representation.
z2frstar	Convert matrix of arcs (and costs) to Forward-Reverse Star.
cpTransf	Transform general convex programs to other forms.
optParamDef	Define optimization parameters in the TOMLAB format (*optParam*).
lpDef	Define optimization parameters in the Optimization Toolbox 1.x format (*optPar*).
mPrint	Print matrix, format: NAME$(i,:)$ $a(i,1)a(i,2)...a(i,n)$.
PrintMatrix	Print matrix with row and column labels.
vPrint	Print vector in rows, format: NAME$(i_1 : i_n)$ $v_{i_1}v_{i_2}...v_{i_n}$.
xPrint	Print vector x, row by row, with format.
xPrinti	Print integer vector x. Calls *xprint*.
xPrinte	Print integer vector x in exponential format. Calls *xprint*.

1.2 How to Solve Optimization Problems Using TOMLAB LDO

This section describes how to use TOMLAB LDO to solve the different type of problems discussed in Section 1.1

1.2.1 How to Solve Linear Programming Problems

The following example shows how the simple LP problem can be solved by direct use of the optimization routines *lpsimp1* and *lpsimp2*:

```
A = [ 1  2
      4  1 ];
b = [ 6 12 ]';
c = [-7 -5]';
meq = 0;
optPar = lpDef;
optPar(13) = meq;
[x_0, B_0, optPar, y] = lpsimp1(A, b, optPar);
[x, B, optPar, y] = lpsimp2(A, b, c, optPar, x_0, B_0);
```

For further illustrations of how to solve linear programming problems see the example files listed in Table 8 and Table 9.

Table 8: Test examples for linear programming.

Function	Description
exinled.m	First simple LP example from a course in Operations Research.
excycle	Menu with cycling examples.
excycle1	The Marshall-Suurballe cycling example. Run both the Bland's cycle preventing rule and the default minimum reduced cost rule and compare results.
excycle2	The Kuhn cycling example.
excycle3	The Beale cycling example.
exKleeM	The Klee-Minty example. Shows that the simplex algorithm with Dantzig's rule visits all vertices.
exfl821	Run exercise 8.21 from Fletcher, Practical methods of Optimization. Illustrates redundancy in constraints.
ex412b4s	Wayne Winston example 4.12 B4, using *lpsimp1* and *lpsimp2*.
expertur	Perturbed both right hand side and objective function for Luenberger 3.12-10,11.
ex6rev17	Wayne Winston chapter 6 Review 17. Simple example of calling the dual simplex solver *lpdual*.
ex611a2	Wayne Winston example 6.11 A2. A simple problem solved with the dual simplex solver *lpdual*.

Some test examples are collected in the file *demoLP* and further described in Table 9.

Table 9: Test examples for linear programming running interior point methods.

Function	Description
exww597	Test of *karmark* and *lpsimp2* on Winston example page 597 and Winston 10.6 Problem A1.
exstrang	Test of *karmark* and *lpsimp2* on Strangs' *nutshell* example.
exkarma	Test of *akarmark*.
exKleeM2	Klee-Minty example solved with *lpkarma* and *karmark*.

1.2.2 How to Solve Transportation Programming Problems

The following is a simple example of a transportation problem

$$s = \begin{pmatrix} 5 \\ 25 \\ 25 \end{pmatrix}, d = \begin{pmatrix} 10 \\ 10 \\ 20 \\ 15 \end{pmatrix}, C = \begin{pmatrix} 6 & 2 & -1 & 0 \\ 4 & 2 & 2 & 3 \\ 3 & 1 & 2 & 1 \end{pmatrix}, \tag{2}$$

where s is the supply vector, d is the demand vector and C is the cost matrix. See *TPsimplx* Section 1.4.19. Solving (2) by use of the routine *TPsimplx* is done by:

```
s = [ 5 25 25 ]';
d = [10 10 20 15 ]';
C = [ 6  2 -1  0
      4  2  2  3
      3  1  2  1 ];

[X, B, optPar, y, C] = TPsimplx(s, d, C)
```

When neither starting base nor starting point is given as input argument *TPsimplx* calls *TPvogel* (using Vogel's approximation method) to find an initial basic feasible solution (bfs). To use another method to find an initial bfs, e.g. the northwest corner method, explicitly call the corresponding routine (*TPnw*) before the call to *TPsimplx*:

```
s = [ 5 25 25 ]';
d = [10 10 20 15 ]';
C = [ 6  2 -1  0
      4  2  2  3
      3  1  2  1 ];

[X_0, B_0] = TPnw(s, d)

[X, B, optPar, y, C] = TPsimplx(s, d, C, X_0, B_0)
```

For further illustrations of how to solve transportation programming problems see the example files listed in Table 10.

Table 10: Test examples for transportation programming.

Function	Description
extp_bfs	Test of the three routines that finds initial basic feasible solution to a TP problem, routines *TPnw*, *TPmc* and *TPvogel*.
exlu119	Luenberger TP page 119. Find initial basis with *TPnw*, *TPmc* and *TPvogel* and run *TPsimplx* for each.
exlu119U	Test unbalanced TP on Luenberger TP page 119, slightly modified. Runs *TPsimplx*.
extp	Runs simple TP example. Find initial basic feasible solution and solve with *TPsimplx*.

1.2.3 How to Solve Network Programming Problems

In TOMLAB LDO there are several routines for network programming problems. Here follows an example of how to solve a shortest path problem. Given the network in Figure 1, where the numbers at each arc represent the distance of the arc, find the shortest path from node 1 to all other nodes. Representing the network with the node-arc incidence matrix A and the cost vector c gives:

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 1 & -1 \\ 0 & 0 & 0 & 0 & -1 & -1 & 0 & 1 \end{pmatrix}, c = \begin{pmatrix} 2 \\ 3 \\ 1 \\ 4 \\ 2 \\ 4 \\ 1 \\ 3 \end{pmatrix} \tag{3}$$

Representing the network with the *forward and reverse star* technique gives:

$$P = \begin{pmatrix} 1 \\ 3 \\ 4 \\ 6 \\ 8 \\ 9 \end{pmatrix}, Z = \begin{pmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 4 \\ 3 & 2 \\ 3 & 5 \\ 4 & 5 \\ 4 & 3 \\ 5 & 4 \end{pmatrix}, c = \begin{pmatrix} 2 \\ 3 \\ 1 \\ 4 \\ 2 \\ 4 \\ 1 \\ 3 \end{pmatrix}, T = \begin{pmatrix} 1 \\ 4 \\ 2 \\ 7 \\ 3 \\ 8 \\ 5 \\ 6 \end{pmatrix}, R = \begin{pmatrix} 1 \\ 1 \\ 3 \\ 5 \\ 7 \\ 9 \end{pmatrix} \tag{4}$$

See *a2frstar* Section 1.5.1 for an explanation of the notation.

Choose *modlabel* to solve this example, see Section 1.4.14, *modlabel* implements a modified label correcting algorithm. First define the incidence matrix A and the cost vector c and call the routine *a2frstar* to convert to a *forward and reverse star* representation (which is used by *modlabel*). Then the actual problem is solved.

```
A = [ 1  1  0  0  0  0  0  0
     -1  0  1 -1  0  0  0  0
      0 -1  0  1  1  0 -1  0
      0  0 -1  0  0  1  1 -1
```

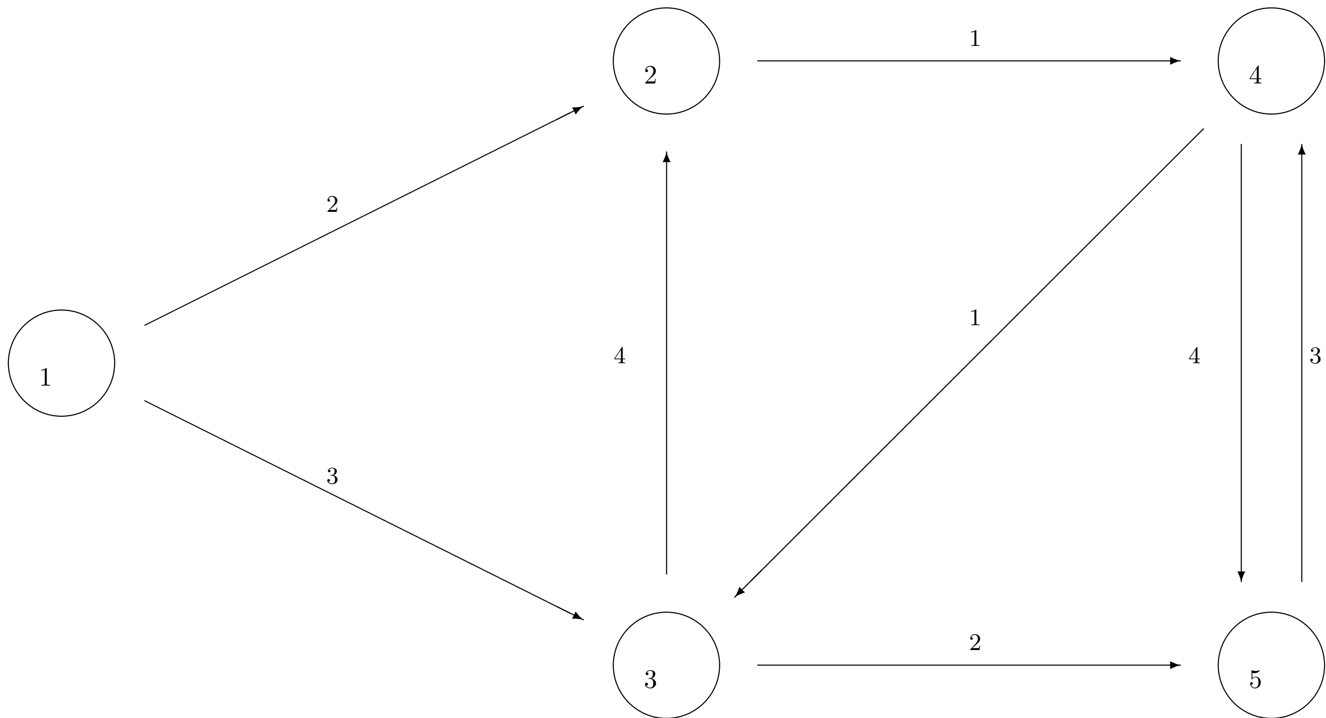

Figure 1: A network example.

```
     0  0  0  0 -1 -1  0  1 ];

C = [ 2  3  1  4  2  4  1  3 ];

[P Z c T R x_U] = a2frstar(A, C);

[pred dist] = modlabel(1,P,Z,c);
```

For further illustrations of how to solve network programming problems see the example files listed in Table 11.

Table 11: Test examples for network programming.

Function	**Description**
exgraph	Testing network routines on simple example.
exflow	Testing several maximum flow examples.
pathflow	Pathological test example for maximum flow problems.
exflow31	Test example N31.
exmcnfp	Minimum Cost Network Flow Problem (MCNFP) example from Ahuja et. al.

1.2.4 How to Solve Integer Programming Problems

The routines originally in TOMLAB LDO for solving integer programming problems were *cutplane*, *mipSolve* and *balas*. Now *cutplane* and *mipSolve* are part of the standard TOMLAB and are called using the TOMLAB format. Examples showing how to use the *balas* routine and the other solvers are listed in Table 12. These examples are all part of the demonstration routine *demoMIP.m*.

Table 12: Test examples for integer programming.

Function	Description
expkorv	Test of *cutplane* and *mipSolve* for example PKorv.
exIP39	Test example I39.
exbalas	Test of 0/1 IP (Balas algorithm) on simple example.

1.2.5 How to Solve Dynamic Programming Problems

In this subsection dynamic programming is illustrated with a simple approach to solve a knapsack problem and an inventory problem. The routines *dpknap* (see Section 1.4.5) and *dpinvent* (Section 1.4.4) are used. The knapsack problem (5) is an example from Holmberg [7] and the inventory problem is an example from Winston [9, page 1013].

$$
\begin{array}{ll}
\max\limits_{u} & f(u) = 7u_1 + u_2 + 4u_3 \\
s/t & 2u_1 + 3u_2 + 2u_3 \leq 4 \\
 & 0 \leq u_1 \leq 1 \\
 & 0 \leq u_2 \leq 1 \\
 & 0 \leq u_3 \leq 2 \\
 & u_j \in \mathbb{N}, j = 1, 2, 3
\end{array}
\tag{5}
$$

Problem (5) will be solved by the following definitions and call:

```
A      = [ 2 3 2 ];
b      =   4;
c      = [ 7 2 4 ];
u_UPP  = [ 1 1 2 ];
PriLev = 0;

[u, f_opt] = dpknap(A, b, c, u_UPP, PriLev);
```

Description of the inventory problem:
A company knows that the demand for its product during each of the next for months will be as follows: month 1, 1 unit; month 2, 3 units; month 3, 2 units; month 4, 4 units. At the beginning of each month, the company must determine how many units should be produced during the current month. During a month in which any units are produced, a setup cost of \$3 is incurred. In addition, there is a variable cost of \$1 for every unit produced. At the end of each month, a holding cost of 50 cents per unit on hand is incurred. Capacity limitations allow a maximum of 5 units to be produced during each month. The size of the company's warehouse restricts the ending inventory for each month to at most 4 units. The company wants to determine a production schedule that will meet all demands on time and will minimize the sum of production and holding costs during the four months. Assume that 0 units are on hand at the beginning of the first month.

The inventory problem described above will be solved by the following definitions and call:

```
d   = [1 3 2 4]';     % Demand. N = 4;
P_s = 3;              % Setup cost $3 if u > 0
P   = ones(5,1);      % Production cost $1/unit in each time step
I_s = 0;              % Zero setup cost for the Inventory
I   = 0.5*ones(5,1); % Inventory cost $0.5/unit in each time step
x_L = 0;              % lower bound on inventory, x
x_U = 4;              % upper bound on inventory, x
x_LAST = [];          % Find best choice of inventory at end
x_S = 0;              % Empty inventory at start
u_L = [0 0 0 0];      % Minimal amount produced in each time step
u_U = [5 5 5 5];      % Maximal amount produced in each time step
```

```
PriLev = 1;

[u, f_opt] = dpinvent(d, P_s, P, I_s, I, u_L, u_U, x_L, x_U, x_S, x_LAST, PriLev);
```

For further illustrations of how to solve dynamic programming problems see the example files listed in Table 13.

Table 13: Test examples for dynamic programming.

Function	Description
exinvent	Test of *dpinvent* on two inventory examples.
exknap	Test of *dpknap* (calls *mipSolve* and *cutplane*) on five knapsack examples.

1.2.6 How to Solve Lagrangian Relaxation Problems

This section shows an example of using Lagrangian relaxation techniques implmented in the routine *ksrelax* to solve an integer programming problem. The problem to be solved, (6), is an example from Fischer [4].

$$
\begin{array}{ll}
\max\limits_{x} & f(x) = 16x_1 + 10x_2 + 4x_4 \\
s/t & 8x_1 + 2x_2 + x_3 + x_4 \leq 10 \\
 & x_1 + x_2 \leq 1 \\
 & x_3 + x_4 \leq 1 \\
 & x_j \in 0/1, j = 1, 2, 3, 4
\end{array}
\tag{6}
$$

```
A = [ 8  2  1  4
      1  1  0  0
      0  0  1  1 ];
b = [10  1  1 ]';
c = [16 10  0  4 ]';
r = 1;                % Do not relax the first constraint
x_UPP = [1 1 1 1]';

[x u f_opt optPar] = ksrelax(A, b, c, r, x_UPP);
```

For further illustrations of how to solve Lagrangian Relaxation problems see the example files listed in Table 14.

Table 14: Test examples for Lagrangian Relaxation.

Function	Description
exrelax	Test of *ksrelax* on LP example from Fischer -85.
exrelax2	Simple example, runs *ksrelax*.
exIP39rx	Test example I39, relaxed. Calls *urelax* and plot.

1.3 Printing Utilities and Print Levels

This section is written for the part of TOMLAB LDO which is not using the same input/output format and is not designed in the same way as the other routines in TOMLAB. Information about printing utilities and print levels for the other routines could be found in the TOMLAB User's guide.

The amount of printing is determined by setting a print level for each routine. This parameter most often has the name *PriLev*.

Normally the zero level ($PriLev = 0$) corresponds to silent mode with no output. The level one corresponds to a result summary and error messages. Level two gives output every iteration and level three displays vectors and matrices. Higher levels give even more printing of debug type. See the help in the actual routine.

The main driver or menu routine called may have a *PriLev* parameter among its input parameters. The routines called from the main routine normally sets the *PriLev* parameter to *optPar(1)*. The vector *optPar* is set to default values by a call to *goptions*. The user may then change any values before calling the main routine. The elements in *optPar* is described giving the command: *help goptions*. For linear programming there is a special initialization routine, *lpDef*, which calls *goptions* and changes some relevant parameters.

There is a wait flag in *optPar*, *optPar(24)*. If this flag is set, the routines uses the pause statement to avoid the output just flushing by.

The TOMLAB LDO routines print large amounts of output if high values for the *PriLev* parameter is set.

For matrices there are two routines, *mPrint* and *PrintMatrix*. The routine *PrintMatrix* prints a matrix with row and column labels. The default is to print the row and column number. The standard row label is eight characters long. The supplied matrix name is printed on the first row, the column label row, if the length of the name is at most eight characters. Otherwise the name is printed on a separate row.

The standard column label is seven characters long, which is the minimum space an element will occupy in the print out. On a 80 column screen, then it is possible to print a maximum of ten elements per row. Independent on the number of rows in the matrix, *PrintMatrix* will first display $A(:, 1 : 10)$, then $A(:, 11 : 20)$ and so on.

The routine *PrintMatrix* tries to be intelligent and avoid decimals when the matrix elements are integers. It determines the maximal positive and minimal negative number to find out if more than the default space is needed. If any element has an absolute value below 10^{-5} (avoiding exact zeros) or if the maximal elements are too big, a switch is made to exponential format. The exponential format uses ten characters, displaying two decimals and therefore seven matrix elements are possible to display on each row.

For large matrices, especially integer matrices, the user might prefer the routine *mPrint*. With this routine a more dense output is possible. All elements in a matrix row is displayed (over several output rows) before next matrix row is printed. A row label with the name of the matrix and the row number is displayed to the left using the Matlab style of syntax.

The default in *mPrint* is to eight characters per element, with two decimals. However, it is easy to change the format and the number of elements displayed. For a binary matrix it is possible to display 36 matrix columns in one 80 column row.

1.4 Optimization Routines in TOMLAB LDO

In the following subsections the optimization routines in TOMLAB LDOwill be described.

1.4.1 akarmark

Purpose
Solve linear programming problems of the form

$$\begin{array}{llll} \min\limits_{x} & f(x) & = & c^T x \\ s/t & Ax & = & b \\ & x & \geq & 0 \end{array}$$

where $x, c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$.

Calling Syntax
[x, optPar, y, x_0] = akarmark(A, b, c, optPar, x_0)

Description of Inputs

A	Constraint matrix.
b	Right hand side vector.
c	Cost vector.
optPar	Optimization parameter vector, see *goptions.m*.
x_0	Starting point.

Description of Outputs

x	Optimal point.
optPar	Optimization parameter vector, see *goptions.m*.
y	Dual parameters.
x_0	Starting point used.

Description
The routine *akarmark* is an implementation of the affine scaling variant of Karmarkar's method as described in Bazaraa [5, pages 411-413]. As the purification algorithm a modified version of the algorithm on page 385 in Bazaraa is used.

Examples
See *exakarma*, *exkarma*, *exkleem2*.

M-files Used
lpDef.m

See Also
lpkarma, *karmark*

1.4.2 balas

Purpose

Solve binary integer linear programming problems.

balas solves problems of the form

$$\begin{array}{cccccl} \min\limits_{x} & f(x) & = & c^T x & \\ s/t & a_i^T x & = & b_i & i = 1, 2, ..., m_{eq} \\ & a_i^T x & \leq & b_i & i = m_{eq} + 1, ..., m \\ & x_j \in 0/1 & & & j = 1, 2, ..., n \end{array}$$

where $c \in \mathbb{Z}^n$, $A \in \mathbb{Z}^{m \times n}$ and $b \in \mathbb{Z}^m$.

Calling Syntax

[x, optPar] = balas(A, b, c, optPar)

Description of Inputs

A	Constraint matrix, integer coefficients.
b	Right hand side vector, integer coefficients.
c	Cost vector, integer coefficients.
$optPar$	Optimization parameter vector, see *goptions.m*.

Description of Outputs

x	Optimal point.
$optPar$	Optimization parameter vector, see *goptions.m*.

Description

The routine *balas* is an implementation of Balas method for binary integer programs restricted to integer coefficients.

Examples

See *exbalas*.

M-files Used

lpDef.m

See Also

mipSolve, *cutplane*

1.4.3 dijkstra

Purpose
Solve the shortest path problem.

Calling Syntax
[pred, dist] = dijkstra(s, P, Z, c)

Description of Inputs

s	The starting node.
p	Pointer vector to start of each node in the matrix Z.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
c	Costs related to the arcs in the matrix Z.

Description of Outputs

pred	$pred(j)$ is the predecessor of node j.
dist	$dist(j)$ is the shortest distance from node s to node j.

Description
dijkstra is a direct implementation of the algorithm DIJKSTRA in [1, pages 250-251] for solving shortest path problems using Dijkstra's algorithm. Dikstra's algorithm belongs to the class of *label setting* methods which are applicable only to networks with nonnegative arc lengths. For solving shortest path problems with arbitrary arc lengths use the routine *labelcor* or *modlabel* which belongs to the class of *label correcting* methods.

Examples
See *exgraph*, *exflow31*.

See Also
labelcor, *modlabel*

Limitations
dijkstra can only solve problems with nonnegative arc lengths.

1.4.4 dpinvent

Purpose
Solve production/inventory problems of the form

$$\begin{array}{llllll}
\min\limits_{u} & f(u) & = & P_s(t) + P(t)^T u(t) + I(t)^T x(t) & & \\
s/t & u_L & \leq & u(t) & \leq & u_U \\
 & x_L & \leq & x(t) & \leq & x_U \\
 & 0 & \leq & u(t) & \leq & x(t) + d(t) \\
 & u_j \in \mathbb{N} & & j = 1, 2, ..., n & & \\
 & x_j \in \mathbb{N} & & j = 1, 2, ..., n & &
\end{array}$$

where $x(t) = x(t-1) + u(t) - d(t)$ and $d \in \mathbb{N}^n$.

Calling Syntax
[u, f_opt, exit] = dpinvent(d, P_s, P, I_s, I, u_L, u_U, x_L, x_U, x_S, x_LAST, PriLev)

Description of Inputs

d	Demand vector.
P_s	Production setup cost.
P	Production cost vector.
I_s	Inventory setup cost.
I	Inventory cost vector.
u_L	Minimal amount produced in each time step.
u_U	Maximal amount produced in each time step.
x_L	Lower bound on inventory.
x_U	Upper bound on inventory.
x_S	Inventory state at start.
x_LAST	Inventory state at finish.
PriLev	Printing level:
	$PriLev = 0$, no output.
	$PriLev = 1$, output of convergence results.
	$PriLev > 1$, output of each iteration.

Description of Outputs

u	Optimal control.
f_opt	Optimal function value.
exit	Exit flag.

Description
dpinvent solves production/inventory problems using a forward recursion dynamic programming technique as described in Winston [9, chap. 20].

Examples
See *exinvent*.

1.4.5 dpknap

Purpose
Solve knapsack problems of the form

$$\begin{array}{cccc} \max\limits_{u} & f(u) & = & c^T u \\ s/t & Au & \leq & b \\ & u & \leq & u_U \\ & u_j \in \mathbb{N} & & j = 1, 2, ..., n \end{array}$$

where $A \in \mathbb{N}^n$, $c \in \mathbb{R}^n$ and $b \in \mathbb{N}$

Calling Syntax
[u, f_opt, exit] = dpknap(A, b, c, u_U, PriLev)

Description of Inputs

A	Weigth vector.
b	Knapsack capacity.
c	Benefit vector.
u_U	Upper bounds on u.
PriLev	Printing level:
	$PriLev = 0$, no output.
	$PriLev = 1$, output of convergence results.
	$PriLev > 1$, output of each iteration.

Description of Outputs

u	Optimal control.
f_opt	Optimal function value.
exit	Exit flag.

Description
dpknap solves knapsack problems using a forward recursion dynamic programming technique as described in [9, chap. 20]. The Lagrangian relaxation routines *ksrelax* and *urelax* call *dpknap* to solve the knapsack subproblems.

Examples
See *exknap*.

1.4.6 karmark

Purpose

Solve linear programming problems of Karmakar's form

$$\begin{array}{cccc} \min\limits_{x} & f(x) & = & c^T x \\ s/t & Ax & = & 0 \\ & \sum\limits_{j=1}^{n} x_j & = & 1 \\ & x & \geq & 0 \end{array}$$

where $x, c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and the following assumptions hold:

- The point $x^{(0)} = \left(\frac{1}{n}, ..., \frac{1}{n}\right)^T$ is feasible.
- The optimal objective value is zero.

Calling Syntax

[x, optPar] = karmark(A, c, optPar)

Description of Inputs

A	Constraint matrix.
c	Cost vector.
optPar	Optimization parameter vector, see *goptions.m*.

Description of Outputs

x	Optimal point.
optPar	Optimization parameter vector, see *goptions.m*.

Description

The routine *karmark* is an implementation of Karmakar's projective algorithm which is of polynomial complexity. The implementation uses the description in Bazaraa [3, page 386]. There is a choice of update, either according to Bazaraa or the rule by Goldfarb and Todd [5, chap. 9]. As the purification algorithm a modified version of the algorithm on page 385 in Bazaraa is used. *karmark* is called by *lpkarma* which transforms linear maximization problems on inequality form into Karmakar's form needed for *karmark*.

Examples

See *exstrang*, *exww597*.

M-files Used

lpDef.m

See Also

lpkarma, *akarmark*

1.4.7 ksrelax

Purpose
Solve integer linear problems of the form

$$\begin{array}{cccc} \max\limits_{x} & f(x) & = & c^T x \\ s/t & Ax & \leq & b \\ & x & \leq & x_U \\ & x_j \in \mathbb{N} & & j = 1, 2, ..., n \end{array}$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{N}^{m \times n}$ and $b \in \mathbb{N}^m$.

Calling Syntax
[x_P, u, f_P, optPar] = ksrelax(A, b, c, r, x_U, optPar)

Description of Inputs

A	Constraint matrix.
b	Right hand side vector.
c	Cost vector.
r	Constraint not to be relaxed.
x_U	Upper bounds on the variables.
optPar	Optimization parameter vector, see *goptions.m*.

Description of Outputs

x_P	Primal solution.
u	Lagrangian multipliers.
f_P	Function value at *x_P*.
optPar	Optimization parameter vector, see *goptions.m*.

Description
The routine *ksrelax* uses Lagrangian Relaxation to solve integer linear programming problems with linear inequality constraints and simple bounds on the variables. The problem is solved by relaxing all but one constraint and then solve a simple knapsack problem as a subproblem in each iteration. The algorithm is based on the presentation in Fisher [4], using subgradient iterations and a simple line search rule. LDO also contains a routine *urelax* which plots the result of each iteration.

Examples
See *exrelax*, *exrelax2*.

M-files Used
lpDef.m, *dpknap.m*

See Also
urelax

1.4.8 labelcor

Purpose
Solve the shortest path problem.

Calling Syntax
[pred, dist] = labelcor(s, P, Z, c)

Description of Inputs

s	The starting node.
p	Pointer vector to start of each node in the matrix Z.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
c	Costs related to the arcs in the matrix Z.

Description of Outputs

pred	$pred(j)$ is the predecessor of node j.
dist	$dist(j)$ is the shortest distance from node s to node j.

Description
The implementation of *labelcor* is based on the algorithm LABEL CORRECTING in [1, page 260] for solving shortest path problems. The algorithm belongs to the class of *label correcting* methods which are applicable to networks with arbitrary arc lengths. *labelcor* requires that the network does not contain any negative directed cycle, i.e. a directed cycle whose arc lengths sum to a negative value.

Examples
See *exgraph*.

See Also
dijkstra, *modlabel*

Limitations
The network must not contain any negative directed cycle.

1.4.9 lpdual

Purpose

Solve linear programming problems when a dual feasible solution is available.

lpdual solves problems of the form

$$\begin{array}{lccll} \min\limits_{x} & f_P(x) & = & c^T x & \\ s/t & a_i^T x & = & b_i & i = 1, 2, ..., m_{eq} \\ & a_i^T x & \leq & b_i & i = m_{eq}+1, ..., m \\ & x & \geq & 0 & \end{array}$$

by rewriting it into standard form and solving the dual problem

$$\begin{array}{lccc} \max\limits_{y} & f_D(y) & = & b^T y \\ s/t & A^T y & \leq & c \\ & y & urs & \end{array}$$

with $x, c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b, y \in \mathbb{R}^m$.

Calling Syntax

[x, y, B, optPar] = lpdual(A, b, c, optPar, B_0, x_0, y_0)

Description of Inputs

A	Constraint matrix.
b	Right hand side vector.
c	Cost vector.
optPar	Optimization parameter vector, see *goptions.m*.
B_0	Logical vector of length n for basic variables at start.
x_0	Starting point, must be dual feasible.
y_0	Dual parameters (Lagrangian multipliers) at *x_0*.

Description of Outputs

x	Optimal point.
y	Dual parameters (Lagrangian multipliers) at the solution.
B	Optimal basic set.
optPar	Optimization parameter vector, see *goptions.m*.

Description

When a dual feasible solution is available, the dual simplex method is possible to use. *lpdual* implements this method using the algorithm in [5, pages 105-106]. There are three rules available for variable selection. Bland's cycling prevention rule is the choice if fear of cycling exist. The other two are variants of minimum reduced cost variable selection, the original Dantzig's rule and one which sorts the variables in increasing order in each step (the default choice).

Examples
See *ex611a2*, *ex6rev17*.

M-files Used
lpDef.m

See Also
lpsimp1, *lpsimp2*

1.4.10 lpkarma

Purpose
Solve linear programming problems of the form

$$\begin{array}{cccc} \max\limits_{x} & f(x) & = & c^T x \\ s/t & Ax & \leq & b \\ & x & \geq & 0 \end{array}$$

where $x, c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$.

Calling Syntax
[x, y, optPar] = lpkarma(A, b, c, optPar)

Description of Inputs

A	Constraint matrix.
b	Right hand side vector.
c	Cost vector.
$optPar$	Optimization parameter vector, see *goptions.m*.

Description of Outputs

x	Optimal point.
y	Dual solution.
$optPar$	Optimization parameter vector, see *goptions.m*.

Description
lpkarma converts a linear maximization problem on inequality form into Karmakar's form and calls *karmark* to solve the transformed problem.

Examples
See *exstrang*, *exww597*.

M-files Used
lpDef.m, *karmark.m*

See Also
karmark, *akarmark*

1.4.11 lpsimp1

Purpose
Find a basic feasible solution to linear programming problems.

lpsimp1 finds a basic feasible solution to problems of the form

$$\begin{array}{cccll} \min\limits_{x} & f(x) & = & c^T x & \\ s/t & a_i^T x & = & b_i & i = 1, 2, ..., m_{eq} \\ & a_i^T x & \leq & b_i & i = m_{eq} + 1, ..., m \\ & x & \geq & 0 & \end{array}$$

where $x, c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m, b \geq 0$.

Calling Syntax
[x, B, optPar, y] = lpsimp1(A, b, optPar)

Description of Inputs

A	Constraint matrix.
b	Right hand side vector.
optPar	Optimization parameter vector, see *goptions.m*.

Description of Outputs

x	Basic feasible solution.
B	Basic set at the solution x.
optPar	Optimization parameter vector, see *goptions.m*.
y	Lagrange multipliers.

Description
The routine *lpsimp1* implements a Phase I Simplex strategy which formulates a LP problem with artificial variables. Slack variables are added to the inequality constraints and artificial variables are added to the equality constraints. The routine uses *lpsimp2* to solve the Phase I problem.

Examples
See *exinled*, *excycle*, *excycle2*, *exKleeM*, *exfl821*, *ex412b4s*.

M-files Used
lpDef.m, *lpsimp2.m*

See Also
lpsimp2

1.4.12 lpsimp2

Purpose
Solve linear programming problems.

lpsimp2 solves problems of the form

$$\begin{array}{llcll} \min\limits_{x} & f(x) & = & c^T x & \\ s/t & a_i^T x & = & b_i & i = 1, 2, ..., m_{eq} \\ & a_i^T x & \leq & b_i & i = m_{eq} + 1, ..., m \\ & x & \geq & 0 & \end{array}$$

where $x, c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$.

Calling Syntax
[x, B, optPar, y] = lpsimp2(A, b, c, optPar, x_0, B_0)

Description of Inputs

A	Constraint matrix.
b	Right hand side vector.
c	Cost vector.
optPar	Optimization parameter vector, see *goptions.m*.
x_0	Starting point, must be a *basic feasible solution*.
B_0	Logical vector of length n for basic variables at start.

Description of Outputs

x	Optimal point.
B	Optimal basic set.
optPar	Optimization parameter vector, see *goptions.m*.
y	Lagrange multipliers.

Description
The routine *lpsimp2* implements the Phase II standard revised Simplex algorithm as formulated in Goldfarb and Todd [5, page 91]. There are three rules available for variable selection. Bland's cycling prevention rule is the choice if fear of cycling exist. The other two are variants of minimum reduced cost variable selection, the original Dantzig's rule and one which sorts the variables in increasing order in each step (the default choice).

Examples
See *exinled*, *excycle*, *excycle1*, *excycle2*, *excycle3*, *exKleeM*, *exfl821*, *ex412b4s*, *expertur*.

M-files Used
lpDef.m

See Also
lpsimp1, *lpdual*

Warnings
No check is done whether the given starting point is feasible or not.

1.4.13 maxflow

Purpose
Solve the maximum flow problem.

Calling Syntax
[max_flow, x] = maxflow(s, t, x_U, P, Z, T, R, PriLev)

Description of Inputs

s	The starting node, the source.
t	The end node, the sink.
P	Pointer vector to start of each node in the matrix Z.
x_U	The capacity on each arc.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
T	Trace vector, points to Z with sorting order Head.
R	Pointer vector in T vector for each node.
PriLev	Printing Level: 0 Silent, 1 Print result (default).

Description of Outputs

max_flow	Maximal flow between node s and node t.
x	The flow on each arc.

Description
maxflow finds the maximum flow between two nodes in a capacitated network using the Ford-Fulkerson augmented path method. The implementation is based on the algorithm description in Luenberger [8, page 144-145].

Examples
See *exflow*, *exflow31*, *pathflow*.

1.4.14 modlabel

Purpose
Solve the shortest path problem.

Calling Syntax
[pred, dist] = modlabel(s, P, Z, c)

Description of Inputs

s	The starting node.
p	Pointer vector to start of each node in the matrix Z.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
c	Costs related to the arcs in the matrix Z.

Description of Outputs

pred	$pred(j)$ is the predecessor of node j.
dist	$dist(j)$ is the shortest distance from node s to node j.

Description
The implementation of *modlabel* is based on the algorithm MODIFIED LABEL CORRECTING in [1, page 262] with the addition of the heuristic rule discussed to improve running time in practice. The rule says: Add *node* to the beginning of $LIST$ if *node* has been in $LIST$ before, otherwise add *node* at the end of $LIST$. The algorithm belongs to the class of *label correcting* methods which are applicable to networks with arbitrary arc lengths. *modlabel* requires that the network does not contain any negative directed cycle, i.e. a directed cycle whose arc lengths sum to a negative value.

Examples
See *exgraph*.

See Also
dijkstra, *labelcor*

Limitations
The network must not contain any negative directed cycle.

1.4.15 NWsimplx

Purpose
Solve the minimum cost network flow problem.

Calling Syntax
[Z, X, xmax, C, S, my, optPar] = NWsimplx(A, b, c, u, optPar)

Description of Inputs

A	Node-arc incidence matrix. A is $m \times n$.
b	Supply/demand vector of length m.
c	Cost vector of length n.
u	Arc capacity vector of length n.
optPar	Optimization parameter vector, see *goptions.m*.

Description of Outputs

Z	Arcs outgoing from the nodes in increasing order.
	$Z(:,1)$ Tail. $Z(:,2)$ Head.
X	Optimal flow.
xmax	Upper bound on the flow.
C	Costs related to the arcs in the matrix Z.
S	Arc status at the solution:
	$S_i = 1$, arc i is in the optimal spanning tree.
	$S_i = 2$, arc i is in L (variable at lower bound).
	$S_i = 3$, arc i is in U (variable at upper bound).
my	Lagrangian multipliers at the solution.
optPar	Optimization parameter vector, see *goptions.m*.

Description
The implementation of the network simplex algorithm in *NWsimplx* is based on the algorithm NETWORK SIMPLEX in Ahuja et al. [2, page 415]. *NWsimplx* uses the forward and reverse star representation technique of the network, described in [2, pages 35-36].

Examples
See *exmcnfp*.

M-files Used
lpDef.m, *a2frstar.m*

1.4.16 qplm

Purpose

Solve equality constrained quadratic programming problems.

qplm solves problems of the form

$$\begin{array}{ll} \min\limits_{x} & f(x) = \frac{1}{2}(x)^T F x + c^T x \\ s/t & Ax = b \end{array}$$

where $x, c \in \mathbb{R}^n$, $F \in \mathbb{R}^{n \times n}$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$.

Calling Syntax

[x, lambda] = qplm(F, c, A, b)

Description of Inputs

F	Constant matrix, the Hessain.
c	Constant vector.
A	Constraint matrix for the linear constraints.
b	Right hand side vector.

Description of Outputs

x	Optimal point.
lambda	Lagrange multipliers.

Description

The routine *qplm* solves a quadratic programming problem, restricted to equality constraints, using the Lagrange method.

See Also

qpBiggs, *qpSolve*, *qpe*

1.4.17 qpe

Purpose
Solve equality constrained quadratic programming problems.

qpe solves problems of the form

$$\begin{array}{ll} \min\limits_{x} & f(x) = \frac{1}{2}(x)^T F x + c^T x \\ s/t & Ax = b \end{array}$$

where $x, c \in \mathbb{R}^n$, $F \in \mathbb{R}^{n \times n}$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$.

Calling Syntax
[x, lambda, QZ, RZ] = qpe(F, c, A, b)

Description of Inputs

F	Constant matrix, the Hessain.
c	Constant vector.
A	Constraint matrix for the linear constraints.
b	Right hand side vector.

Description of Outputs

x	Optimal point.
lambda	Lagrange multipliers.
QZ	The matrix Q in the QR-decomposition of F.
RZ	The matrix R in the QR-decomposition of F.

Description
The routine *qpe* solves a quadratic programming problem, restricted to equality constraints, using a null space method.

See Also
qpBiggs, *qpSolve*, *qplm*

1.4.18 salesman

Purpose

Solve the symmetric travelling salesman problem.

Calling Syntax

[Tour, f_tour, OneTree, f_tree, w_max, my_max, optPar] =
salesman(C, Zin, Zout, my, f_BestTour, optPar)

Description of Inputs

C	Cost matrix of dimension $n \times n$ where $C_{ij} = C_{ji}$ is the cost of arc (i, j). If there are no arc between node i and node j then set $C_{ij} = C_{ji} = \infty$. It must hold that $C_{ii} = NaN$.
Zin	List of arcs forced in.
Zout	List of arcs forced out.
my	Lagrange multipliers.
f_BestTour	Cost (total distance) of a known tour.
optPar	Optimization parameter vector, see *goptions.m*.

Description of Outputs

Tour	Arc list of the best tour found.
f_tour	Cost (total distance) of the best tour found.
OneTree	Arc list of the best 1-tree found.
f_tree	Cost of the best 1-tree found.
w_max	Best dual objective.
my_max	Lagrange multipliers at *w_max*.
optPar	Optimization parameter vector, see *goptions.m*.

Description

The routine *salesman* is an implementation of an algorithm by Held and Karp [6] which solves the symmetric travelling salesman problem using Lagrangian Relaxation. The dual problem is solved using a subgradient method with the step length given by the Polyak rule II. The primal problem is to find a 1-tree. Here the routine *mintree* is called to get a minimum spanning tree. With this method there is no guarantee that an optimal tour is found, i.e. a zero duality gap can not be guaranteed. To ensure convergence, *salesman* could be used as a subroutine in a Branch and Bound algorithm, see *travelng* which calls *salesman*.

Examples

See *ulyss16*.

M-files Used

lpDef.m, *mintree.m*

See Also

travelng

1.4.19 TPsimplx

Purpose
Solve transportation programming problems.

TPsimplx solves problems of the form

$$
\begin{array}{llcll}
\min\limits_{x} & f(x) & = & \sum\limits_{i}^{m}\sum\limits_{j}^{n} c_{ij}x_{ij} & \\
s/t & \sum\limits_{j}^{n} x_{ij} & = & s_i & i = 1,2,...,m \\
 & \sum\limits_{i}^{n} x_{ij} & = & d_j & j = 1,2,...,n \\
 & x & \geq & 0 &
\end{array}
$$

where $x, c \in \mathbb{R}^{m \times n}$, $s \in \mathbb{R}^m$ and $d \in \mathbb{R}^n$.

Calling Syntax
[X, B, optPar, y, C] = TPsimplx(s, d, C, X, B, optPar, Penalty)

Description of Inputs

s	Supply vector, length m.
d	Demand vector, length n.
C	The cost matrix of linear objective function coefficients.
X	Basic Feasible Solution matrix.
B	Index (i, j) of basis found.

optPar Optimization parameter vector, see *goptions.m*. Fields used by *TPsimplx* are:

optPar(1)	Print level:	
	0	No output
	> 0	Convergence results
	> 1	Output every iteration
	> 2	Output in each step in the simplex algorithm
optPar(14)	Maximum number of iterations. Default: $max(10 * dim(s) * dim(d), 100)$.	
optPar(24)	Wait flag, pause each iteration if > 0.	

Penalty If the problem is unbalanced with $\sum\limits_{i}^{m} s_i < \sum\limits_{j}^{n} d_j$, a dummy supply point is added with cost vector *Penalty*. If the length of *Penalty* $< n$ then the value of the first element in *Penalty* is used for the whole added cost vector. Default: Computed as $10\max(C_{ij})$.

Description of Outputs

X	Solution matrix.
B	Optimal set. Index (i, j) of the optimal basis found.

optPar	Optimization parameter vector, see *goptions.m*.			
	optPar(8)	$\hat{f} = c^T x$ at optimum (or last iterate if no convergence)		
			0	OK.
	optPar(28)	Exit flag:	1	Maximum number of iterations reached. No solution found.
			2	Unbounded feasible region.
y	Lagrange multipliers at solution.			
C	The cost matrix, changed if the problem is unbalanced.			

Description

The routine *TPsimplx* is an implementation of the Transportation Simplex method described in Luenberger [8, chap 5.4]. In LDO, three routines to find a starting basic feasible solution for the transportation problem are included; the Northwest Corner method (*TPnw*), the Minimum Cost method (*TPmc*) and Vogel's approximation method (*TPvogel*). If calling *TPsimplx* without giving a starting point then Vogel's method is used to find a starting basic feasible solution.

Examples

See *extp_bfs*, *exlu119*, *exlu119U*, *extp*.

M-files Used

TPvogel.m

See Also

TPmc, *TPnw*, *TPvogel*

Warnings

No check is done whether the given starting point is feasible or not.

1.4.20 urelax

Purpose

Solve integer linear problems of the form

$$
\begin{array}{cccc}
\max\limits_{x} & f(x) & = & c^T x \\
s/t & Ax & \leq & b \\
 & x & \leq & x_U \\
 & x_j \in \mathbb{N} & & j = 1, 2, ..., n
\end{array}
$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{N}^{m \times n}$ and $b \in \mathbb{N}^m$.

Calling Syntax

[x_P, u, f_P] = urelax(u_max, A, b, c, r, x_U, optPar)

Description of Inputs

u_max	Upper bounds on u.
A	Constraint matrix.
b	Right hand side vector.
c	Cost vector.
r	Constraint not to be relaxed.
x_U	Upper bounds on the variables.
optPar	Optimization parameter vector, see *goptions.m*.

Description of Outputs

x_P	Primal solution.
u	Lagrangian multipliers.
f_P	Function value at *x_P*.

Description

The routine *urelax* is a simple example of the use of Lagrangian Relaxation to solve integer linear programming problems. The problem is solved by relaxing all but one constraint and then solve a simple knapsack problem as a subproblem in each iteration. *urelax* plots the result of each iteration. LDO also contains a more sophisticated routine, *ksrelax*, for solving problems of this type.

Examples

See *exip39rx*.

M-files Used

dpknap.m

See Also

ksrelax

1.5 Optimization Subfunction Utilities in TOMLAB LDO

In the following subsections the optimization subfunction utilities in TOMLAB LDO will be described.

1.5.1 a2frstar

Purpose
Convert a node-arc incidence matrix representation of a network to the forward and reverse star data storage representation.

Calling Syntax
[P, Z, c, T, R, u] = a2frstar(A, C, U)

Description of Inputs

A	The node-arc incidence matrix. A is $m \times n$, where m is the number of arcs and n is the number of nodes.
C	Cost for each arc, n-vector.
U	Upper bounds on flow (optional).

Description of Outputs

P	Pointer vector to start of each node in the matrix Z.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
c	Costs related to the arcs in the matrix Z.
T	Trace vector, points to Z with sorting order Head.
R	Rewerse pointer vector in T for each node.
u	Upper bounds on flow if U is given as input, else infinity.

Description
The routine *a2frstar* converts a node-arc incidence matrix representation of a network to the forward and reverse star data storage representation as described in Ahuja et.al. [2, pages 35-36].

Examples
See *exflow*, *exflow31*, *exgraph*, *pathflow*.

1.5.2 gsearch

Purpose
Find all nodes in a network which is reachable from a given source node.

Calling Syntax
[pred, mark] = gsearch(s, P, Z, c)

Description of Inputs

s	The starting node.
P	Pointer vector to start of each node in the matrix Z.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
c	Costs related to the arcs in the matrix Z.

Description of Outputs

pred	$pred(j)$ = Predecessor of node j.
mark	If $mark(j) = 1$ the node is reachable from node s.

Description
gsearch is searching for all nodes in a network which is reachable from the given source node s. The implementation is a variation of the Algorithm SEARCH in [1, pages 231-233]. The algorithm uses a depth-first search which means that it creates a path as long as possible and backs up one node to initiate a new probe when it can mark no new nodes from the tip of the path. A stack approach is used where nodes are selected from the front and added to the front.

Examples
See *exgraph*.

See Also
gsearchq

1.5.3 gsearchq

Purpose
Find all nodes in a network which is reachable from a given source node.

Calling Syntax
[pred, mark] = gsearchq(s, P, Z, c)

Description of Inputs

s	The starting node.
P	Pointer vector to start of each node in the matrix Z.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
c	Costs related to the arcs in the matrix Z.

Description of Outputs

pred	$pred(j)$ = Predecessor of node j.
mark	If $mark(j) = 1$ the node is reachable from node s.

Description
gsearchq is searching for all nodes in a network which is reachable from the given source node s. The implementation is a variation of the Algorithm SEARCH in [1, pages 231-233]. The algorithm uses a breadth-first search which means that it visits the nodes in order of increasing distance from s. The distance being the minimum number of arcs in a directed path from s. A queue approach is used where nodes are selected from the front and added to the rear.

Examples
See *exgraph*.

See Also
gsearch

1.5.4 mintree

Purpose
Find the minimum spanning tree of an undirected graph.

Calling Syntax
[Z_tree, cost] = mintree(C, Zin, Zout)

Description of Inputs

C	Cost matrix of dimension $n \times n$ where $C_{ij} = C_{ji}$ is the cost of arc (i, j). If there are no arc between node i and node j then set $C_{ij} = C_{ji} = \infty$. It must hold that $C_{ii} = NaN$.
Zin	List of arcs which should be forced to be included in Z_tree.
Zout	List of arcs which should not be allowed to be included in Z_tree (could also be given as NaN in C).

Description of Outputs

Z_tree	List of arcs in the minimum spanning tree.
cost	The total cost.

Description
mintree is an implementation of Kruskal's algorithm for finding a minimal spanning tree of an undirected graph. The implementation follows the algorithm description in [2, page 520-521]. It is possible to give as input, a list of those arcs which should be forced to be included in the tree as well as a list of those arcs which should not be allowed to be included in the tree. *mintree* is called by *salesman*.

1.5.5 TPmc

Purpose
Find a basic feasible solution to the Transportation Problem.

Calling Syntax
[X,B] = TPmc(s, d, C)

Description of Inputs

s	Supply vector of length m.
d	Demand vector of length n.
C	The cost matrix of linear objective function coefficients.

Description of Outputs

X	Basic feasible solution matrix.
B	Index (i, j) of the basis found.

Description
TPmc is an implementation of the Minimum Cost method for finding a basic feasible solution to the transportation problem. The implementation of this algorithm follows the algorithm description in Winston [9, chap. 7.2].

Examples
See *extp_bfs*, *exlu119*, *exlu119U*, *extp*.

See Also
TPnw, *TPvogel*, *TPsimplx*

1.5.6 TPnw

Purpose
Find a basic feasible solution to the Transportation Problem.

Calling Syntax
[X, B] = TPnw(s, d)

Description of Inputs

s	Supply vector of length m.
d	Demand vector of length n.

Description of Outputs

X	Basic feasible solution matrix.
B	Index (i, j) of the basis found.

Description
TPnw is an implementation of the Northwest Corner method for finding a basic feasible solution to the transportation problem. The implementation of this algorithm follows the algorithm description in Winston [9, chap. 7.2].

Examples
See *extp_bfs*, *exlu119*, *exlu119U*, *extp*.

See Also
TPmc, *TPvogel*, *TPsimplx*

1.5.7 TPvogel

Purpose

Find a basic feasible solution to the Transportation Problem.

Calling Syntax

[X, B] = TPvogel(s, d, C, PriLev)

Description of Inputs

s	Supply vector of length m.
d	Demand vector of length n.
C	The cost matrix of linear objective function coefficients.
$PriLev$	If $PriLev > 0$, the matrix X is displayed in each iteration. If $PriLev > 1$, pause in each iteration. Default: $PriLev = 0$.

Description of Outputs

X	Basic feasible solution matrix.
B	Index (i, j) of the basis found.

Description

TPvogel is an implementation of Vogel's method for finding a basic feasible solution to the transportation problem. The implementation of this algorithm follows the algorithm description in Winston [9, chap. 7.2].

Examples

See *extp_bfs*, *exlu119*, *exlu119U*, *extp*.

See Also

TPmc, *TPnw*, *TPsimplx*

1.5.8 z2frstar

Purpose
Convert a table of arcs and corresponding costs in a network to the forward and reverse star data storage representation.

Calling Syntax
[P, Z, c, T, R, u] = z2frstar(Z, C, U)

Description of Inputs

Z	A table with arcs (i, j). Z is $n \times 2$, where n is the number of arcs. The number of nodes m is set equal to the greatest element in Z.
C	Cost for each arc, n-vector.
U	Upper bounds on flow (optional).

Description of Outputs

P	Pointer vector to start of each node in the matrix Z.
Z	Arcs outgoing from the nodes in increasing order. $Z(:, 1)$ Tail. $Z(:, 2)$ Head.
c	Costs related to the arcs in the matrix Z.
T	Trace vector, points to Z with sorting order Head.
R	Rewerse pointer vector in T for each node.
u	Upper bounds on flow if U is given as input, else infinity.

Description
The routine *z2frstar* converts a table of arcs and corresponding costs in a network to the forward and reverse star data storage representation as described in Ahuja et.al. [2, pages 35-36].

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Network flows. In G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd, editors, *Optimization*, volume 1 of *Handbooks in Operations Research and Management Science*. Elsevier/North Holland, Amsterdam, The Netherlands, 1989.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice-Hall Inc., Kanpur and Cambridge, 1993.

[3] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali. *Linear Programming and Network Flows*. John Wiley and Sons, New York, 2nd edition, 1990.

[4] Marshall L. Fisher. An Application Oriented Guide to Lagrangian Relaxation. *Interfaces 15:2*, pages 10–21, March-April 1985.

[5] D. Goldfarb and M. J. Todd. Linear programming. In G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd, editors, *Optimization*, volume 1 of *Handbooks in Operations Research and Management Science*. Elsevier/North Holland, Amsterdam, The Netherlands, 1989.

[6] Michael Held and Richard M. Karp. The Traveling-Salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 1:6–25, 1971.

[7] Kaj Holmberg. Heltalsprogrammering och dynamisk programmering och flöden i nätverk och kombinatorisk optimering. Technical report, Division of Optimization Theory, Linköping University, Linköping, Sweden, 1988-1993.

[8] David G. Luenberger. *Linear and Nonlinear Programming*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2nd edition, 1984.

[9] Wayne L. Winston. *Operations Research: Applications and Algorithms*. International Thomson Publishing, Duxbury Press, Belmont, California, 3rd edition, 1994.