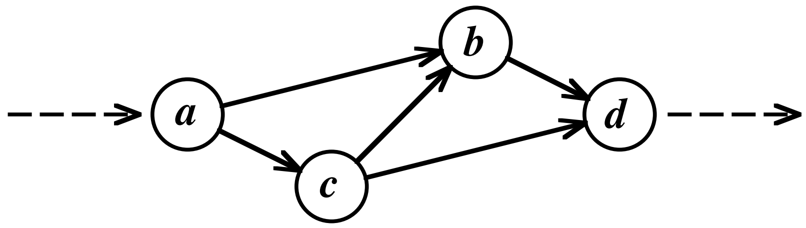
IE D50-1, Fall 1989–90
Mathematical Programming I
Prof. Fourer

ASSIGNMENT #4

Due Wednesday, December 6

1 This problem starts out with a very simple "traffic flow" network,



and some data (to be explained) for each of the "roads" in the network:

From	To	*Time*	*Capacity*	*Sensitivity*
a	b	5.0	10	0.1
a	c	1.0	30	0.9
c	b	2.0	10	0.9
b	d	1.0	30	0.9
c	d	5.0	10	0.1

To be specific, we will imagine that the times are in minutes, the capacities are in cars per minute, and the sensitivities are in minutes per (car per minute).

The following statements can be used to represent networks of this kind in AMPL:

```
set inters;   # road intersections

param S symbolic in inters;   # origin of flow
param T symbolic in inters;   # destination of flow

set roads within {i in inters, j in inters: i != T and j != S};

param time {roads} > 0;
param cap {roads} > 0;
param sens {roads} > 0;
```

For the diagrammed network, the origin `S` is at intersection a, and the destination `T` is at intersection d. There are five members of the set of roads: `(a,b)`, `(a,c)`, `(c,b)`, `(b,d)` and `(c,d)`.

a: What is the shortest path, in minutes, from node `S` to node `T`? Explain why the following AMPL model is guaranteed to find the answer to this question:

```
var x {roads} >= 0;   # x[i,j] = 1 if (i,j) is on shortest path
                      # x[i,j] = 0 otherwise

minimize path_length:  sum {(i,j) in roads} time[i,j] * x[i,j];

subject to flow {i in inters: i != S and i != T}:
    sum {(i,j) in roads} x[i,j] = sum {(j,i) in roads} x[j,i];

subject to source:  sum {(S,j) in roads} x[S,j] = 1;
```

Add the data for the simple example above, and verify that this linear program gives the shortest path as 4 minutes along $a \to c \to b \to d$. Save the model file on your disk as `A:TRAFFIC1.MOD`.

b: What is the maximum traffic flow, in cars per minute, that the network can sustain? Devise a linear program that will allow you to answer this question for any traffic network. Use AMPL to verify that, for the small example above, the answer is 30 cars per minute, with 10 on `(a,b)`, `(c,b)` and `(c,d)`, and 20 on `(a,c)` and `(b,d)`. Save the model file on your disk as `A:TRAFFIC2.MOD`.

c: Question (a) above was concerned only with the speed of traffic, and question (b) only with the volume of traffic. In reality, these quantities are interrelated. As the traffic volume on a road increases from zero, the time required to travel the road also increases.

Travel time increases only moderately when there are just a few cars, but it increases very rapidly as the traffic approaches capacity. Thus a nonlinear function is needed to model this phenomenon. Let us define `t[i,j]`, the travel time on road `(i,j)`, by the following constraints:

```
var x {roads} >= 0;   # cars per minute entering road (i,j)
var t {roads} >= 0;   # travel time for road (i,j)

subject to travel_time {(i,j) in roads}:
    t[i,j] = time[i,j] + (sens[i,j]*x[i,j]) / (1-x[i,j]/cap[i,j]);

subject to capacity {(i,j) in roads}:  x[i,j] <= .999 * cap[i,j];
```

You can confirm that the travel time on `(i,j)` is close to `time[i,j]` when the road is lightly used, but goes to infinity as the use approaches `cap[i,j]` cars per minute. The magnitude of `sens[i,j]` controls the rate at which travel time increases, as more cars try to enter the road. (The bound in the capacity constraint is `.999 * cap[i,j]`, rather than just `cap[i,j]`, because `t[i,j]` would evaluate to infinity if `x[i,j]` were allowed to equal `cap[i,j]`. This is no problem in theory, but it creates a great deal of difficulty for the nonlinear programming algorithm.)

Suppose that we want to analyze how the network can best handle some number of cars per minute. The objective is to minimize average travel time from `S` to `T`:

```
param through > 0;   # cars per minute using the network

minimize avg_time:
    (sum {(i,j) in roads} t[i,j] * x[i,j]) / through;
```

The nonlinear expression `t[i,j] * x[i,j]` is travel minutes on road `(i,j)` times cars per minute entering the road—hence, the number of cars on road `(i,j)`. The summation in the objective thus gives the total cars in the entire system. Dividing by the number of cars per minute getting through, we have the average number of minutes for each car.

To guarantee the postulated throughput, the nonlinear program also needs the following linear constraints:

```
subject to flow {i in inters: i != S and i != T}:
    sum {(i,j) in roads} x[i,j] = sum {(j,i) in roads} x[j,i];

subject to source:  sum {(S,j) in roads} x[S,j] = through;
```

Use AMPL to show that, for the example given above, a throughput of 4.0 cars per minute is optimally managed by sending half the cars along $a \rightarrow b \rightarrow d$ and half along $a \rightarrow c \rightarrow d$, giving an average travel time of about 8.18 minutes. Save the model file on your disk as `A:TRAFFIC3.MOD`.

By trying other values, develop a complete graph of minimum average travel time versus throughput. Also, keep a record of which travel routes are used in the optimal solutions for different throughputs, and summarize this information on your graph.

Find and explain as many relationships as you can between the information in your graph and the solutions from parts (a) and (b).

d: The model in (c) assumes that you can make the cars' drivers take certain routes. For example, in the optimal solution for a throughput of 4.0, no drivers are allowed to "cut through" from c to b.

What would happen if instead all drivers could take whatever route they pleased? Observation has shown that, in such a case, the traffic tends to reach a *stable* solution in which no route has a travel time less than the average. The optimal solution for a throughput of 4.0 is not stable, since—as you can verify from the solution listing—the travel time on $a \rightarrow c \rightarrow b \rightarrow d$ is only about 7.86 minutes; some drivers would try to cut through if they were permitted.

To find a stable solution using AMPL, we have to add some data specifying the possible routes from `S` to `T`:

```
param choices integer > 0;   # number of routes

set route {1..choices} within roads;
```

Here `route` is an indexed collection of sets; for each `r` in `1..choices`, the expression `route[r]` denotes a different subset of roads that together form a route from `S` to `T`. Using this data, the stability conditions may be ensured by one additional collection of constraints:

```
subject to stability {r in 1..choices}:
    sum {(i,j) in route[r]} t[i,j] >=
        (sum {(i,j) in roads} t[i,j] * x[i,j]) / through;
```

Show that, in the stable solution for a throughput of 4.0, a little more than 5% of the drivers cut through, and the average travel time increases to about

8.27 minutes. Thus traffic would have been faster if the road from c to b had never been built! (This phenomenon is known as Braess's paradox, in honor of a traffic analyst who noticed that when a certain link was added to Munich's road system, traffic seemed to get worse.) Save the model file on your disk as `A:TRAFFIC4.MOD`.

By trying other values, develop a complete graph of the stable travel time, and indicate for which throughputs it is greater than the optimal time.

e: Suppose now that there is an additional winding road, directly from a to d, with travel time 5 minutes, capacity 10, and sensitivity 1.5.

Working with the models developed above, write an analysis of the consequences of making this change as a function of the throughput value.

2 You will be given a Pascal program called `SIMPLEX.PAS`, which reads linear programs that are expressed in the following simple form:

```
 3   5

 1.1  1.3  1.7  1.6  1.0

 2.3  3.6  4.5  7.4  4.0   3.6
 4.5  8.7  1.8  6.2  4.5   0.7
-0.9  2.2 -6.5  6.5  4.5   8.6

 1   3   4
```

The first line specifies the numbers of constraints and variables, respectively, and the second line lists the coefficients of the variables in the objective function. Each subsequent lines gives the coefficients of one constraint, plus a right-hand side constant as the last value. The final line lists the indices of the variables that are to be in the initial basis.

Included in `SIMPLEX.PAS` are subroutines to factorize a basis matrix as the product of simpler matrices, and subroutines that can use such a factorization to solve linear equation systems in the basis and in its transpose. Also included are the steps needed to read the input, to set up and factor the initial basis, and to initialize various arrays relating to the basis.

Copy `SIMPLEX.PAS` to your disk, and complete it so that it carries out "phase II" of the simplex method to *minimize* the objective function. Be sure that your program does at least all of the following at each simplex iteration:

- Performs all the calculations required by the algorithm, including solution of $B^T\pi = c_B$ and $By_B = a_{N_p}$, determination of reduced costs, and application of the minimum-ratio test.
- Checks for optimality or unboundedness.
- Updates the basic solution, $\bar{x}_B$.
- Prints one line of output that gives the iteration number, the indices of the entering and leaving variables, the reduced cost, the minimum ratio, and the value of the objective function.

When the algorithm terminates, print an appropriate message. If optimality is indicated, then also print the optimal objective value *and* the optimal solution.

To help you get your program working, three test problems have been provided along with the program. They are in the files named `TEST3-1.DAT`, `TEST3-2.DAT` and `TEST3-3.DAT`. *The correct optimal value of the objective function is indicated at the end of each file.*

Hint #1: You will find a constant named `opt_tol` defined at the start of the file `SIMPLEX.PAS`. Your program should not regard a reduced cost to be acceptably negative unless it is less than `-opt_tol`.

Hint #2: The arrays declared at the beginning of `SIMPLEX.PAS` are all the ones necessary for a short and efficient implementation of the simplex method.

Hint #3: To get columns of numbers to line up in your output, you will need to include some formatting information in your `WRITE` or `WRITELN` statements. For example,

`WRITE (i:4)`	prints the value of integer variable `i` in a field four characters wide; if `i` is 12, for instance, this prints two blanks, a 1 and a 2
`WRITE (x[j]:8:2)`	prints the value of real variable `x[j]` in a field eight characters wide, with two decimal places; if `x[j]` is 12.5, for instance, this prints three blanks followed by 12.50

Hand in a disk containing the completed program along with the written part of the assignment.

Questions or comments to:

Prof. Robert Fourer or Prof. Collette Coullard
Department of Industrial Engineering and Management Sciences
Northwestern University
Evanston, IL 60208-3119, U.S.A.

708-491-3151
708-491-8005 (fax)

`4er@iems.nwu.edu` or `coullard@iems.nwu.edu`