Implementing Linear Programming-Based Network Revenue Management Models

1. Linear programming in JAVA and QSOpt

Implementing linear programming-based network revenue management models requires successively solving many linear programs as the departure time of the flights approaches and the capacities on the flight legs are consumed. Consequently, to be able to test these models in a simulation environment, it is imperative that the linear programs are solved through a programming language.

We use QSOpt to solve linear programs in JAVA. QSOpt is a linear programming software developed by researchers at Georgia Institute of Technology and it provides construction, manipulation and reporting facilities for linear programs. Details on QSOpt can be found at `www.isye.gatech.edu/~wcook/qsopt/`.

We begin by an example that solves the linear program

$$\begin{aligned} \max \quad & 4\,x + y + 3\,z \\ \text{subject to} \quad & x + 4\,y \leq 1 \\ & 3\,x - y + z \leq 3 \\ & 0 \leq x \leq 10 \\ & 0 \leq y \leq 20 \\ & 0 \leq z \end{aligned}$$

There are many strategies to construct a linear program in QSOpt, but we pick one particular strategy and stick with it. Our strategy first creates "empty" constraints that only include the "sense" and the right hand sides. In particular, we leave the constraint coefficients of the decision variables unspecified. After this, we consider each decision variable one by one and specify its objective function and constraint coefficients.

It is also useful to emphasize at this point that QSOpt thinks that there are only two constraints in the linear program above (as opposed to five). The last three constraints are simple bounds on the decision variables and are not treated as "serious" constraints by QSOpt. One could, of course, specify these bounds on the decision variables as "serious" constraints as well but this makes the linear program a bit more difficult to specify.

The following piece of code can be used to solve the linear program above in JAVA.

```
package RevenueManagement.QSOptDemo;

import qs.*;

public class QSOptDemo
{
    public static void main ( String[] args )
    {
```

```
try
{
    Problem lp = new Problem( "Test Problem" );
    /* maximize the objective */
    lp.change_objsense ( QS.MAX );
    /* build right hand coefficients */
    double[] rhs = new double[ 2 ];
    rhs[ 0 ] = 1.0; rhs[ 1 ] = 3.0;
    /* build sense of the constraints */
    char sense[] = new char[ 2 ];
    sense[ 0 ] = 'L'; sense[ 1 ] = 'L';
    /* the size of this array is equal to the number of variables but it is all zeros */
    int[] varcnt = new int[ 3 ];
    /* add the constraints to the problem */
    /* the first parameter is the number of constraints we are adding */
    lp.add_rows( 2 , varcnt , null , null , null , rhs , sense , null);

    /* start adding the objective coefficients of the variables into the linear program */
    /* we have 3 variables, so the first component of the arrays have dimension 3 */
    /* keeps the number of nonzero coefficients of each variable */
    int[][] matcnt_var = new int[ 3 ][];
    int[][] matbeg_var = new int[ 3 ][];
    /* this array keeps constraint index of the nonzero coefficients of each variable */
    int[][] matind_var = new int[ 3 ][];
    /* keeps constraint coefficients of the nonzero coefficients of each variable */
    double[][] matval_var = new double[ 3 ][];
    /* the following arrays are the objective function value, upper bound etc. */
    double[][] obj_var = new double[ 3 ][ 1 ];
    double[][] lower_var = new double[ 3 ][ 1 ];
    double[][] upper_var = new double[ 3 ][ 1 ];
    String[][] name_var = new String[ 3 ][ 1 ];

    /* create x variable */
    matcnt_var[ 0 ] = new int[ 1 ];
    matcnt_var[ 0 ][ 0 ] = 2;
    matbeg_var[ 0 ] = new int[ 1 ];
    matbeg_var[ 0 ][ 0 ] = 0;
    /* note x variable has two nonzero coefficients in the constraints */
    matind_var[ 0 ] = new int[ 2 ];
    matind_var[ 0 ][ 0 ] = 0; matind_var[ 0 ][ 1 ] = 1;
    matval_var[ 0 ] = new double[ 2 ];
    matval_var[ 0 ][ 0 ] = 1.0; matval_var[ 0 ][ 1 ] = 3.0;
    /* create y variable */
    matcnt_var[ 1 ] = new int[ 1 ];
    matcnt_var[ 1 ][ 0 ] = 2;
    matbeg_var[ 1 ] = new int[ 1 ];
    matbeg_var[ 1 ][ 0 ] = 0;
    /* note y variable has two nonzero coefficients in the constraints */
    matind_var[ 1 ] = new int[ 2 ];
    matind_var[ 1 ][ 0 ] = 0; matind_var[ 1 ][ 1 ] = 1;
    matval_var[ 1 ] = new double[ 2 ];
    matval_var[ 1 ][ 0 ] = 4.0; matval_var[ 1 ][ 1 ] = -1.0;
    /* create z variable */
    matcnt_var[ 2 ] = new int[ 1 ];
    matcnt_var[ 2 ][ 0 ] = 1;
```

```
            matbeg_var[ 2 ] = new int[ 1 ];
            matbeg_var[ 2 ][ 0 ] = 0;
            /* note z variable has one nonzero coefficients in the constraints */
            matind_var[ 2 ] = new int[ 1 ];
            matind_var[ 2 ][ 0 ] = 1;
            matval_var[ 2 ] = new double[ 1 ];
            matval_var[ 2 ][ 0 ] = 1.0;

            /* construct objective coefficients, lower bounds, upper bounds and names */
            obj_var[ 0 ][ 0 ] = 4.0; obj_var[ 1 ][ 0 ] = 1.0; obj_var[ 2 ][ 0 ] = 3.0;
            lower_var[ 0 ][ 0 ] = 0.0; lower_var[ 1 ][ 0 ] = 0.0; lower_var[ 2 ][ 0 ] = 0.0;
            upper_var[ 0 ][ 0 ] = 10.0; upper_var[ 1 ][ 0 ] = 20.0;
                                        upper_var[ 2 ][ 0 ] = QS.MAXDOUBLE;
            name_var[ 0 ][ 0 ] = "x"; name_var[ 1 ][ 0 ] = "y"; name_var[ 2 ][ 0 ] = "z";

            /* add variables into problem */
            for ( int i = 0 ; i < 3 ; i ++ )
            {
                lp.add_cols( 1 , matcnt_var[ i ] , matbeg_var[ i ] , matind_var[ i ] ,
                   matval_var[ i ] , obj_var[ i ] , lower_var[ i ] , upper_var[ i ] , name_var[ i ] );
            }

            /* solve the problem */
            lp.opt_dual();
        }
        catch ( Exception e )
        {
            throw new Error ( e.getMessage() + " " + e.getClass() );
        }
    }
}
```

The following piece of code can be used to manipulate the linear program that we constructed.

```
    /* write the problem to a file */
    lp.write_lp ( "c:\\test.lp" );

    /* get the solution - first create an array with length 3 since we have 3 variables */
    double[] x = new double[ 3 ];
    lp.get_x_array( x );
    for ( int i = 0 ; i < 3 ; i ++ )
        System.out.println( x[ i ] );

    /* get the dual variables - first create an array with length 2 since we have 2 constraints */
    double[] pi = new double[ 2 ];
    lp.get_pi_array( pi );
    for ( int i = 0 ; i < 2 ; i ++ )
        System.out.println( pi[ i ] );
```

It is instructive to sketch the arrays that we constructed so that we can see how JAVA arrays are used to construct the coefficients of the linear program.

2. Building network revenue management models in JAVA

We begin by constructing JAVA objects that are useful to model network revenue management problems. The following piece of code gives a sketch of a class that keeps the locations in the airline network.

```
package RevenueManagement.MultiLeg.General;

public class LocationKeeper
{
    String[] locations;
    int noLocations;
    int size;
    static LocationKeeper instance;

    public LocationKeeper ( int noLocations_ )
    {
        noLocations = noLocations_;
        locations = new String[ noLocations ];
        size = 0;
        instance = this;
    }

    public void addLocation ( String location )
    {
        locations[ size ] = location;
        size ++;
    }

    public String getLocation ( int index )
    {
        return locations[ index ];
    }

    public int getLocationIndex ( String location )
    {
        for ( int i = 0 ; i < size ; i ++ )
        {
            if ( locations[ i ].equals( location ) )
                return i;
        }
        throw new Error ( "cannot find location " + location );
    }

    public String[] getLocations()
    {...}
    public int getNoLocations()
    {...}
    public static LocationKeeper instance()
    {...}
}
```

The locations are kept as strings and each location has an integer index associated with it. Since the locations have integer indices associated with them, we can use an array to keep the flight legs associated with each origin-destination pair. Before illustrating how we keep the flight legs, we sketch a class that represents a flight leg.

```
package RevenueManagement.MultiLeg.General;

public class Flight
{
    String origin;
    String destination;
    int capacity;

    public Flight( String origin_ , String destination_ , int capacity_ )
    {
        origin = origin_;
        destination = destination_;
        capacity = capacity_;
    }

    public String getOrigin()
    {...}
    public String getDestination()
    {...}
    public int getCapacity()
    {...}
}
```

The following class simply holds the flight legs. This class is useful to refer to a flight leg in the airline network or to deal with a large number of flight legs in an efficient manner.

```
package RevenueManagement.MultiLeg.General;

/* each flight is indexed by an integer, this is useful for LP formulation */
public class FlightKeeper
{
    /* indices of the following array are origin and destination index of the flight */
    Flight[][] flights;
    int[][] flightIndices;
    int size;
    static FlightKeeper instance;

    public FlightKeeper( int noLocations )
    {
        flights = new Flight[ noLocations ][ noLocations ];
        flightIndices = new int[ noLocations ][ noLocations ];
        instance = this;
    }
```

```
    public void addFlight ( Flight flight )
    {
        /* get origin and destination indices for the flight */
        int orIndex = LocationKeeper.instance().getLocationIndex( flight.getOrigin() );
        int desIndex = LocationKeeper.instance().getLocationIndex( flight.getDestination() );
        /* add the flight in the appropriate place */
        flights[ orIndex ][ desIndex ] = flight;
        flightIndices[ orIndex ][ desIndex ] = size;
        size ++;
    }

    /* returns the flight between a particular origin destination */
    /* returns null is there is no such flight */
    public Flight getFlight ( String origin , String destination )
    {
        int orIndex = LocationKeeper.instance().getLocationIndex( origin );
        int desIndex = LocationKeeper.instance().getLocationIndex( destination );
        return flights[ orIndex ][ desIndex ];
    }

    /* returns the flight with a particular index */
    public Flight getFlight ( int index )
    {...}
    /* returns the index of the flight between a particular origin destination */
    public int getFlightIndex ( String origin , String destination )
    {...}
    /* returns the index of a flight  */
    public int getFlightIndex ( Flight flight )
    {...}
    public int getNoFlights()
    {...}
    public static FlightKeeper instance()
    {...}
}
```

The flight keeper class has various methods to look up flight legs. For example, we can easily retrieve the flight leg associated with a particular origin-destination pair. Another important thing is that we have an integer index associated with each flight leg. This will be useful when formulating the network revenue management problem as a linear program. Since this linear program has one constraint for each flight leg, we use the integer index for the flight leg as the index for the corresponding constraint. The flight keeper class also provides several ways to look up the integer index for a flight leg.

The following class represents an itinerary. Since there can be many flight legs included in an itinerary, this class includes an array of flight legs. This array represents the flight legs in the itinerary.

```
package RevenueManagement.MultiLeg.General;
```

```
public class Itin
{
    /* no of flight legs in the itinerary */
    int noFlights;
    /* origin and destination for the itin */
    String origin;
    String destination;
    /* flights in the itinerary */
    Flight[] flights;
    /* fare class for itinerary */
    int fareClass;
    /* fare for the itinerary */
    double fare;
    /* counts the number of flights already added to the itinerary */
    int size;

    public Itin ( int noFlights_ , String origin_,  String destination_ ,
                                                    int fareClass_ , double fare_ )
    {
        noFlights = noFlights_;
        origin = origin_;
        destination = destination_;
        flights = new Flight[ noFlights ];
        fareClass = fareClass_;
        fare = fare_;
        size = 0;
    }

    public void addFlight ( Flight flight )
    {
        flights[ size ] = flight;
        size ++;
    }

    public Flight[] getFlights()
    {
        return flights;
    }

    public int getFareClass()
    {...}
    public double getFare()
    {...}
    public String getOrigin()
    {...}
    public String getDestination()
    {...}
}
```

We also construct a class to hold the itineraries. Similar to the flight keeper class, this class includes an array that keeps the itineraries by origin, destination and fare class, and provides several methods to look up itineraries. Furthermore, we associate an integer index with each

itinerary and this becomes useful when formulating the network revenue management problem as a linear program. Since this linear program has one decision variable for each itinerary, we use the integer index for the itinerary as the index for the corresponding decision variable. The itinerary keeper class also provides several ways to look up the integer index of an itinerary. The following piece of code sketches the itinerary keeper.

```
package RevenueManagement.MultiLeg.General;

/* each itin is indexed by an integer, this is useful for the linear programming formulation */
public class ItinKeeper
{
    /* array's indices are origin destination and fare class */
    Itin[][][] itins;
    int[][][] itinIndices;
    int size;
    static ItinKeeper instance;

    public ItinKeeper( int noLocations , int noFareClasses )
    {...}
    public void addItin ( Itin itin )
    {...}
    /* returns the itin between a particular origin destination and fare class */
    public Itin getItin ( String origin , String destination , int fareClass )
    {...}
    /* returns the itin with a particular index */
    public Itin getItin ( int index )
    {...}
    /* returns the index of the itin between a particular origin destination and fare class*/
    public int getItinIndex ( String origin , String destination , int fareClass )
    {...}
    /* returns the index of an itinerary */
    public int getItinIndex( Itin itin )
    {...}
    public int getNoItins()
    {...}
    public static ItinKeeper instance()
    {...}
}
```

To specify the arrival process for the itinerary requests, we need the probability of having a request for a particular itinerary at a particular time period. The following class keeps these probabilities.

```
package RevenueManagement.MultiLeg.General;

public class ProbabilityKeeper
{
```

```
    /* indices of the array are no of time periods and itineraries */
    double[][] probabilities;
    /* useful for generating samples of the itinerary requests */
    double[][] cdf;
    static ProbabilityKeeper instance;

    public ProbabilityKeeper ( int noTimePeriods , int noItins )
    {
        probabilities = new double[ noTimePeriods ][ noItins ];
        instance = this;
    }

    public void addProbability ( int time , Itin itin , double p )
    {
        int itinIndex = ItinKeeper.instance().getItinIndex( itin );
        probabilities[ time ][ itinIndex ] = p;
    }

    public double getProbability ( int time , Itin itin )
    {...}

    /* constructs the cdf of itinerary arrivals from the pdf */
    public void contructCDF()
    {
        cdf = new double[ GlobalParamsKeeper.instance().getNoTimePeriods() ]
                                            [ ItinKeeper.instance().getNoItins() ];
        for ( int t = 0 ; t < GlobalParamsKeeper.instance().getNoTimePeriods() ; t ++ )
        {
            cdf[ t ][ 0 ] = probabilities[ t ][ 0 ];
            for ( int i = 1 ; i < ItinKeeper.instance().getNoItins() ; i ++ )
            {
                cdf[ t ][ i ] = cdf[ t ][ i - 1 ] + probabilities[ t ][ i ];
            }
        }
    }

    /* generates a sample of the itinerary arrival at time t */
    public Itin sample ( int time )
    {
        double r = Math.random();
        for ( int i = 0 ; i < cdf[ time ].length ; i ++ )
        {
            if ( r < cdf[ time ][ i ] )
            {
                return ItinKeeper.instance().getItin( i );
            }
        }
        ...
    }

    public static ProbabilityKeeper instance()
    {...}
}
```

We adopt the following format to store problem parameters.

```
# out file
c:\\huseyin\\research\\RevenueManagement\\MultiLeg\\Data\\out.txt

# number of time periods
100

# locations - simple list
# first line is number of locations
3
100
101
102

# flights - from to capacity
# first line is number of flights
4
101 100 23
102 100 7
100 101 18
100 102 23

# itineraries - from, to, class, fare, no flights in itinerary, list of flights in itinerary
# first line is number of itineraries and no fare classes
12 2
100 101 0 24.0 1 [ 100 101 ]
100 101 1 192.0 1 [ 100 101 ]
100 102 0 34.0 1 [ 100 102 ]
100 102 1 272.0 1 [ 100 102 ]
101 100 0 24.0 1 [ 101 100 ]
101 100 1 192.0 1 [ 101 100 ]
101 102 0 53.0 2 [ 101 100 ] [ 100 102 ]
101 102 1 424.0 2 [ 101 100 ] [ 100 102 ]
102 100 0 34.0 1 [ 102 100 ]
102 100 1 272.0 1 [ 102 100 ]
102 101 0 53.0 2 [ 102 100 ] [ 100 101 ]
102 101 1 424.0 2 [ 102 100 ] [ 100 101 ]

# probabilities - time period itinerary probability
0 [ 100 101 0 ] 0.26 [ 100 101 1 ] 0.0 [ 100 102 0 ] 0.26 [ 100 102 1 ] 0.0 [ 101 100 0 ] 0.15...
1 [ 100 101 0 ] 0.26 [ 100 101 1 ] 0.0 [ 100 102 0 ] 0.26 [ 100 102 1 ] 0.0 [ 101 100 0 ] 0.15...
2 [ 100 101 0 ] 0.26 [ 100 101 1 ] 0.0 [ 100 102 0 ] 0.26 [ 100 102 1 ] 0.0 [ 101 100 0 ] 0.15...
3 [ 100 101 0 ] 0.26 [ 100 101 1 ] 0.0 [ 100 102 0 ] 0.26 [ 100 102 1 ] 0.0 [ 101 100 0 ] 0.15...
4 [ 100 101 0 ] 0.26 [ 100 101 1 ] 0.0 [ 100 102 0 ] 0.26 [ 100 102 1 ] 0.0 [ 101 100 0 ] 0.15...
5 [ 100 101 0 ] 0.26 [ 100 101 1 ] 0.0 [ 100 102 0 ] 0.26 [ 100 102 1 ] 0.0 [ 101 100 0 ] 0.15...
6 [ 100 101 0 ] 0.26 [ 100 101 1 ] 0.0 [ 100 102 0 ] 0.26 [ 100 102 1 ] 0.0 [ 101 100 0 ] 0.15...
7 [ 100 101 0 ] 0.26 [ 100 101 1 ] 0.0 [ 100 102 0 ] 0.26 [ 100 102 1 ] 0.0 [ 101 100 0 ] 0.15...
...
```

The final two portions of this file format are interesting. When listing the itineraries, we

explicitly list what flight legs are included in each itinerary. The final portion of the file format has one line for each time period. For each time period, we list each itinerary and the probability of having a request for that itinerary.

We construct a straightforward class for reading the problem parameters. We also construct a global parameters keeper class that keeps a few global parameters. For our problem class, the only global parameter is the number of time periods in the planning horizon. The global parameters keeper class also keeps a print writer that becomes useful when we want to write some data to an output file.

3. Building the linear program in JAVA

The linear program for the network revenue management problem can be written as

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} f_j \, w_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} \, w_j \leq x_{it} \qquad i = 1, \ldots, m \\ & 0 \leq w_j \leq \sum_{t'=t}^{\tau} p_{jt'} \quad j = 1, \ldots, n, \end{aligned}$$

where $\{x_{it} : i = 1, \ldots, m\}$ are the remaining capacities on the flight legs at time period t and $\{\sum_{t'=t}^{\tau} p_{jt'} : j = 1, \ldots, n\}$ are the expected number of itinerary requests during the time interval $\{t, \ldots, \tau\}$. We solve this linear program at each time period as the capacities on the flight legs are consumed and the expected numbers of itinerary requests decrease. If we write this linear program in matrix notation as

$$\begin{aligned} \max \quad & f \, w \\ \text{subject to} \quad & A \, w \leq x_t \\ & 0 \leq w \leq P_t, \end{aligned}$$

then it is easy to see that the matrix A and the vector f do not change over time. In our implementation, we construct these elements of the linear program once at the beginning and do not change them at all. On the other hand, the vectors x_t and P_t change as the system evolves and we have to reconstruct these elements of the linear program at each time period. Another important point from the perspective of implementation is that the second set of constraints are not "serious" constraints and they can simply be constructed as upper bounds. Consequently, the number of decision variables in the linear program is equal to the number of itineraries and the number of constraints is equal to the number of flight legs.

We begin by a simple class that keeps the remaining capacity on each flight leg. We note that these capacities appear in the right side of the first set of constraints in the linear program.

```
package RevenueManagement.MultiLeg.BidPriceLP;
```

```
import RevenueManagement.MultiLeg.General.*;

public class CapacityKeeper
{
    /* indices of the array are the origin and destinations of the flights */
    int[][] capacities;
    static CapacityKeeper instance;

    public CapacityKeeper()
    {
        capacities = new int[ LocationKeeper.instance().getNoLocations() ]
                                            [ LocationKeeper.instance().getNoLocations() ];
        instance = this;
    }

    /* set capacities so that they reflect initial flight capacities */
    public void initialize()
    {
        for ( int o = 0 ; o < LocationKeeper.instance().getNoLocations() ; o ++ )
        {
            for ( int d = 0 ; d < LocationKeeper.instance().getNoLocations() ; d ++ )
            {
                String origin = LocationKeeper.instance().getLocation( o );
                String destination = LocationKeeper.instance().getLocation( d );
                Flight flight = FlightKeeper.instance().getFlight( origin , destination );
                if ( flight != null )
                {
                    capacities[ o ][ d ] = flight.getCapacity();
                }
            }
        }
    }

    public int getCapacity ( Flight flight )
    {...}

    /* after selling the itinerary itin, adjust the flight leg capacities */
    public void adjust ( Itin itin )
    {
        Flight[] flights = itin.getFlights();
        for ( int f = 0 ; f < flights.length ; f ++ )
        {
            Flight flight = flights[ f ];
            int origin = LocationKeeper.instance().getLocationIndex( flight.getOrigin() );
            int destination = LocationKeeper.instance().getLocationIndex( flight.getDestination() );
            capacities[ origin ][ destination ] --;
        }
    }

    public static CapacityKeeper instance()
    {...}
}
```

The following piece of code gives the constructor for the linear program.

```
public BidPriceLP()
{
    /* number of variables is equal to the number of itineraries */
    rmatcnt_var = new int[ ItinKeeper.instance().getNoItins() ][ 1 ];
    rmatbeg_var = new int[ ItinKeeper.instance().getNoItins() ][ 1 ];
    /* we do not yet know how many constraints each itinerary decision variable appears */
    rmatind_var = new int[ ItinKeeper.instance().getNoItins() ][];
    rmatval_var = new double[ ItinKeeper.instance().getNoItins() ][];
    obj_var = new double[ ItinKeeper.instance().getNoItins() ][ 1 ];
    lower_var = new double[ ItinKeeper.instance().getNoItins() ][ 1 ];
    upper_var = new double[ ItinKeeper.instance().getNoItins() ][ 1 ];
    name_var = new String[ ItinKeeper.instance().getNoItins() ][ 1 ];
    /* number of constraints is equal to the number of flight legs */
    rhs_const = new double[ FlightKeeper.instance().getNoFlights() ];
    sense_const = new char[ FlightKeeper.instance().getNoFlights() ];
    /* length of this array has to be equal to the number of variables - contents do not matter */
    varcount_const = new int[ ItinKeeper.instance().getNoItins() ];

    /* generate the constraint elements of A matrix and objective function*/
    generateConstraintMatrix();
    /* generate the sense of the constraints */
    generateConstraints();
}
```

We fill the gaps in the constructor above by the following two methods.

```
public void generateConstraintMatrix()
{
    /* we use itinerary indices as the indices of decision variables in LP */
    for ( int i = 0 ; i < ItinKeeper.instance().getNoItins() ; i ++ )
    {
        Itin itin = ItinKeeper.instance().getItin( i );
        Flight[] flights = itin.getFlights();
        /* the constraints this variable appears in depends on the flights in this itinerary */
        rmatcnt_var[ i ][ 0 ] = flights.length;
        rmatbeg_var[ i ][ 0 ] = 0;
        rmatind_var[ i ] = new int[ flights.length ];
        rmatval_var[ i ] = new double[ flights.length ];
        for ( int f = 0 ; f < flights.length ; f ++ )
        {
            Flight flight = flights[ f ];
            int flindx = FlightKeeper.instance().getFlightIndex( flight );
            rmatind_var[ i ][ f ] = flindx;
            rmatval_var[ i ][ f ] = 1.0;
        }
        obj_var[ i ][ 0 ] = itin.getFare();
```

```
        lower_var[ i ][ 0 ] = 0;
        upper_var[ i ][ 0 ] = 0;
        name_var[ i ][ 0 ] = "I_" + itin.getOrigin() + "_"
                                  + itin.getDestination() + "_" + itin.getFareClass();
    }
}
```

```
public void generateConstraints()
{
    /* we use flight indices as the indices of the constraints LP */
    for ( int f = 0 ; f < FlightKeeper.instance().getNoFlights() ; f ++ )
    {
        Flight flight = FlightKeeper.instance().getFlight( f );
        /* set to zero momentarily, this will change later */
        rhs_const[ f ] = 0;
        sense_const[ f ] = 'L';
    }
}
```

At each time period, we reconstruct the right sides of the constraints. After this, we construct the linear program with the right hand sides (that we just constructed) and with the objective function and constraint matrix coefficients (that we constructed at the very beginning).

```
public void construct ( int time )
{
    lp = new Problem( "Bid Price LP" );
    lp.change_objsense( QS.MAX );
    /* construct right hand sides of constraints by looking at the capacity keeper */
    for ( int f = 0 ; f < FlightKeeper.instance().getNoFlights() ; f ++ )
    {
        Flight flight = FlightKeeper.instance().getFlight( f );
        rhs_const[ f ] = CapacityKeeper.instance().getCapacity( flight );
    }
    /* construct upper bounds of variables by looking at probability keeper */
    for ( int i = 0 ; i < ItinKeeper.instance().getNoItins() ; i ++ )
    {
        Itin itin = ItinKeeper.instance().getItin( i );
        double p = 0;
        for ( int t = time ; t < GlobalParamsKeeper.instance().getNoTimePeriods() ; t ++ )
        {
            p += ProbabilityKeeper.instance().getProbability( t , itin );
        }
        upper_var[ i ][ 0 ] = p;
    }
    /* add constraints to LP */
```

```
    lp.add_rows( FlightKeeper.instance().getNoFlights() , varcount_const , null , null , null ,
                                                          rhs_const , sense_const , null);
    /* add variables related to each itinerary one-by-one */
    for ( int i = 0 ; i < ItinKeeper.instance().getNoItins() ; i ++ )
    {
        lp.add_cols( 1 , rmatcnt_var[ i ] , rmatbeg_var[ i ] , rmatind_var[ i ] , rmatval_var[ i ] ,
                         obj_var[ i ] , lower_var[ i ] , upper_var[ i ] , name_var[ i ] );
    }
}
```

---

4. Simulating the performance of the policy characterized by the linear program

We use the following class to simulate the performance of the policy characterized by the linear program. At each time period, we first sample an itinerary request. If we do not have enough capacity to serve this itinerary request, then there is no decision to make. Otherwise, we solve the linear program using the remaining leg capacities and the expected numbers of itinerary requests. Using the dual variables associated with the capacity availability constraints as the bid-prices, we compute the sum of the bid-prices of the flights legs included in the requested itinerary. If the revenue from the requested itinerary exceeds the sum of the bid-prices, then we accept the itinerary request and adjust the remaining leg capacities. The following piece of code sketches the simulator.

```
public void simulate()
{
    for ( int it = 0 ; it < noIterations ; it ++ )
    {
        double revenue = 0;
        /* initialize capacity keeper so that it reflects initial capacities of flights */
        CapacityKeeper.instance().initialize();
        /* loop over all time periods */
        for ( int time = 0 ; time < GlobalParamsKeeper.instance().getNoTimePeriods() ; time ++ )
        {
            /* sample an itinerary request */
            Itin sample = ProbabilityKeeper.instance().sample( time );
            /* check if we have available capacity to serve the itinerary */
            boolean available = checkCapacities( sample );
            /* if we have capacity, then solve the bid price LP to get bid prices */
            if ( available )
            {
                BidPriceLP.instance().construct( time );
                BidPriceLP.instance().solve();
                BidPriceLP.instance().getLP().get_pi_array( duals );
                /* compute the sum of the bid prices in the itinerary */
                double bidPrices = addBidPrices( duals , sample );
                /* if fare exceeds sum of the bid prices, then sell */
                if ( sample.getFare() >= bidPrices )
                {
                    revenue += sample.getFare();
                    CapacityKeeper.instance().adjust( sample );
                }
```

```
            }
        }
    }
}
```

We fill the gaps in the piece of code above by the following two methods.

```
public boolean checkCapacities ( Itin itin )
{
    Flight[] flights = itin.getFlights();
    for ( int f = 0 ; f < flights.length ; f ++ )
    {
        if ( CapacityKeeper.instance().getCapacity( flights[ f ] ) <= 0 )
            return false;
    }
    return true;
}
```

```
public double addBidPrices( double[] duals , Itin itin )
{
    double sum = 0;
    Flight[] flights = itin.getFlights();
    for ( int f = 0 ; f < flights.length ; f ++ )
    {
        /* get constraint index corresponding to this flight */
        int flindex = FlightKeeper.instance().getFlightIndex( flights[ f ] );
        sum -= duals[ flindex ];
    }
    return sum;
}
```
