NAME

Class SPNetwork − An interface for reading in and using data found in the stochastic processing network (and all subsets of the Stochastic Processing Network) input file.

SYNOPSIS

```
#include<SPNetwork.h>

struct MatrixEntry;
{
   int i;
   int j;
   double val;
}

SPNetwork( const string filename );
SPNetwork( TaggedValues& input );
SPNetwork(const SPNetwork& copyee);

void printNetwork();
bool usesProbRouting();
bool usesDetRouting();
bool usingPools();
bool isStochastic();
int getClasses();
int getServers();
double getTotalEventRate();
double* getArrivalRates();
double* getHoldingCosts();
int* getServerForClass();
int* getserverPoolSizes();
int* getSuccessors();
bool* getNonidling();
MatrixEntry* getServiceRates( int& size );
int getServiceRateLength();
MatrixEntry* getTransitionProbabilities( int& size );
int getTransitionProbLength();
bool isFeasible( int** u );
bool isStateFeasible( int** u, int x[] );
std::vector<int**> getFeasibleActions()
```

DESCRIPTION

Provides a consistent interface through which to read network input files describing Stochastic Processing Networks (thus: SPNetwork), and allows access to the data within the files. Can calculate feasible actions (which are network−dependent) and determine if a given action/state pair is feasible.

There are only two constructors. One allows programmers to simply specify the filename from which they plan on reading the network, the other takes a reference to a **TaggedValues** and allows the programmer to access values other than those needed by **SPNetwork** through the **TaggedValues** interface.

The struct **MatrixEntry** is an important part of **SPNetwork** and is public because the interface is needed to be able to access the elements of the transition probability matrix and the service rate matrix. Both matrices are sparse, and so **MatrixEntry** allows the program to iterate over ONLY the non-zero elements of either matrix. This usually cuts down on the programs' runtimes.

MatrixEntry.i and **MatrixEntry.j** are indices into a matrix and **MatrixEntry.val** is the value at that location. For example:

```
MatrixEntry xij;
xij.i = 1;
xij.j = 3;
xij.val = 3.14;
```

creates a single entry in a matrix that represents the data given by the following assignment `x[i][j] = 3.14;` It is assumed that all indices which do not have **MatrixEntry**s have a value of zero. For more on using the **MatrixEntry** struct see the examples below.

printNetwork() does just what its name suggests. Calling this will print out a summary of the information gathered and stored by **SPNetwork**.

The methods returning booleans: **usesProbRouting**(), **usesDetRouting**(), **usingPools**(), and **isStochastic**(), return values determined while building the object and so provide instant access to facts about the network.

The methods **getClasses**() and, **getServers**() return the number of classes and servers that are part of the network. These serve as the lengths of most of the arrays returned by other acecssors. The values are those read in from the input file, either by this class or by the **TaggedValues** class.

getArrivalRates(), **getHoldingCosts**(), **getServerPoolSizes**(), **getSuccessors**(), **getServiceRates**(*int&* size) and, **getTransitionProbabilities**(*int&* size) all return the information described by their names. Note: **getServiceRates** and **getTransitionProbabilities** return **MatrixEntry** arrays instead of doubly indexed arrays. Because only non-zero entries in these matrices are recorded, the size variable is passed in, and before returning is changed to match the length of the **MatrixEntry** array. (Therefore: index these arrays from 0-->[size-1]). **getSuccessors**() will fail if called on a network defined with probabilities instead of successors (see SPNetwork.5 for more details on defining input files).

getServiceRateLength() and, **getTransitionProbLength()** are accessors for the number of non-zeroes in their respective matrices in case you need access to the size of the matrix somewhere where the size variable initially passed into **get...**(*int&* size) is out of scope.

getTotalEventRate() is calculated from the above data. It is equal to the sum of the maximum service rates for each server. **getServerForClass**() is also a derived attribute. Dependent on mu, this will return the server responsible for the class you subscript it by. For example in a series-2 line it would return (1,2). In a Rybko-Stolyar network it would return (1,2,2,1). **getServerForClass**() will fail if called on a network in which any class can be served by more than one server.

Finally, **getNonidling**() is a boolean array which allows access to the booleans which are true if that class is not allowed to idle unless it has no jobs available to it.

Besides the above functions there are three functions which are designed to make iterating over all the possible actions and states easier. The function **isFeasible**(*int*** u) allows you to pick an action and then check if it is feasible or not based on the network. **isStateFeasible**(*int*** u, *int* x[]) checks to see whether a particular action/state combination is possible. VERY importantly, these functions require the actions to have been allocated using the function **allocateArray**() (**allocateArray(3)**). Finally, **getFeasibleActions**() returns a vector of all the possible actions so that code needs to calculate which ones are possible only once. This also allows code to be much cleaner because it keeps the complicated feasiblity tests encapsulated. Non-idling classes are not considered in determining feasibility.

EXAMPLES

MatrixEntry is an important class to use. The following show two different ways of multiplying mu (the completion rates) by its corresponding u (action):

```
//Allocation and assignment of u
...
//Allocation and assignment of mu
...
double sumUTimesMu = 0.0;
for( int i = 0; i <= numberOfServers; i++ )
{
```

```
        for( int j = 0; j <= numberOfClasses; j++ )
        {
            sumUTimesMu += u[i] * mu[i];
        }
    }
```

As you can see, this could waste a lot of time, especially if the number of servers and classes is relatively large. The following does the same job, but uses the sparce matrix:

```
//Allocation and assignment of u
...
//Initialization of SPNetwork with "iniFileName" (a string)
SPNetwork network(iniFileName);

//Allocation and assignment of sparce mu using MatrixEntry:
int muSize = 0;
MatrixEntry* mu = network.getServiceRates(muSize);

//Now muSize is set to the number of entries within sparceMu
double sumUTimesMu = 0.0;
for( int n = 0; n <= muSize; n++ )
{
    sumUTimesMu += mu[n].val * u[mu[n].i][mu[n].j];
}
```

While it is much more cumbersome to use **MatrixEntry** it makes sparse matrices take up a lot less time to iterate over and occupy less space in memory.

isFeasible(*int*** u) also deserves a moment of explanation. If you produce a policy array (sized: numebr of servers by number of classes) you can hand it to this method which will check to make sure it's legal for this network.

SEE ALSO

SPNetwork.5

AUTHOR

The class described here was originally implemented by Christopher Pfohl in 2009. It has been modified and improved by Dr. Jonathan Senning.

Copyright © 2009 Christopher W. Pfohl, Department of Mathematics and Computer Science, Gordon College, 255 Grapevine Road, Wenham MA, 01984